REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
09-05-2011

2. REPORT TYPE

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Sinkhole Avoidance Routing in Wireless Sensor Networks

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Stephenson, Andrew John

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Naval Academy
Annapolis, MD 21402

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Trident Scholar Report no. 401 (2011)

12. DISTRIBUTION/AVAILABILITY STATEMENT
This document has been approved for public release; its distribution is UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Wireless sensor networks, or WSNs, are an emerging commercial technology that may have practical applications on the modern battlefield. A wireless sensor network consists of individual sensor nodes that work cooperatively to collect and communicate environmental data. In a surveillance role, a WSN could be deployed across a geographic area of interest, allowing military commanders to monitor enemy troop positions and movements. Wireless sensor networks have enormous potential as an information gathering tool, but they also present many unique challenges to security engineers. An adversary can easily capture and tamper with one of the many unguarded sensor nodes to disrupt or significantly degrade the quality of surveillance that the WSN provides. This project examined potential attacks against WSNs and developed a modified routing protocol that increases the overall data integrity and reliability of wireless sensor networks.

15. SUBJECT TERMS
wireless sensor networks, sinkhole attack, routing protocol

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
				51	19b. TELEPHONE NUMBER *(Include area code)*

Sinkhole Avoidance Routing in Wireless Sensor Networks

MIDN 1/C Andrew J. Stephenson
Trident Research Project
Information Technology Major

Adviser: Dr. Eric Harder

May 9, 2011

The purpose of computing is insight, not numbers.
–Richard Hamming

Abstract

Wireless sensor networks, or WSNs, are an emerging commercial technology that may have practical applications on the modern battlefield. A wireless sensor network consists of individual sensor nodes that work cooperatively to collect and communicate environmental data. In a surveillance role, a WSN could be deployed across a geographic area of interest, allowing military commanders to monitor enemy troop positions and movements. Wireless sensor networks have enormous potential as an information gathering tool, but they also present many unique challenges to security engineers. An adversary can easily capture and tamper with one of the many unguarded sensor nodes to disrupt or significantly degrade the quality of surveillance that the WSN provides. This project examined potential attacks against WSNs and developed a modified routing protocol that increases the overall data integrity and reliability of wireless sensor networks.

Due to battery limitations of individual sensor nodes, many WSN protocols seek to conserve power by simplifying computations and reducing the number of radio transmissions required for communication. These practices allow the WSN to have a longer life expectancy; however, such protocols are easy targets for enemy exploitation. In what is known as a sinkhole attack, a comprised sensor node is maliciously used to alter the wireless mesh of a sensor network for the purpose of disrupting the logical

flow of information across the network. The purpose of this project is to minimize the disruption from such an attack. We have proposed modifications to an existing tree based routing protocol so that it attempts to avoid sinkholes and increase the overall data throughput of the network by sacrificing some of the networks transmission efficiency. The efficacy of the project's proposed sinkhole avoidance strategy is also supported through the use of software based WSN network simulations.

Acknowledgements

I would like to acknowledge the following people for their help in support of my Trident research project. Without them this project would not have been possible:
Dr. Harder – For his guidance, wisdom, and patience as my project advisor.
CDR Vincent, USN – For his encouragement to pursue a WSN research project.
Dr. Brown – For his advice and suggestions throughout the progression of my project.
Dr. Dillner – For her input and encouragement during my project review.
Dr. Wick – For his support of the Trident Scholar program.
The Trident Committee – For allowing me the opportunity to perform a rewarding undergraduate research project on behalf of the United States Naval Academy.

Contents

1 Motivation For Research

In business, a misinformed decision may lead to falling stock prices. In war, a misinformed decision may lead to death. A warrior does not deal in dollars, euros, or yen. He or she deals in the currency of human life. Command decisions are made based on known information. Accurate and timely information can lead a commander to make the correct decision under the severest of time constraints. In a modern war zone, seconds can dictate the difference between success and failure. An emerging technology – wireless sensor networks – may some day provide reliable battlefield information to commanders in real time, reducing risk and saving lives.

2 Introduction

A wireless sensor network, or WSN, refers to a group of small battery powered sensors. An individual sensor, commonly referred to as a node, consists of five major parts: a processor, digital memory, a radio, a sensor suite, and a battery. Additionally, a sensor node can be fitted with actuators that allow it to generate power, move about its environment, or perform some specific task. At its most basic level, a single WSN node is designed to be a sensor. Typical sensor suites are capable of detecting changes in light, sound, temperature, pressure, or acceleration. More sophisticated sensors can be used to detect seismic activity, chemicals, or even radiation [1]. Wireless sensor networks can be used in a variety of peaceful applications, for example: equipment monitoring in industrial facilities, pollution monitoring outside of power plants, or allergen monitoring inside of hospitals. Wireless sensor networks also have the potential for many military applications. Hundreds or even thousands of wireless sensors could be dropped from aircraft and spread over a wide geographic area. These sensors would be able to set up a surveillance network used to monitor enemy troop and equipment movement. In addition, a wireless sensor network could be strategically deployed by special forces near points of interest. Considering the small size of sensor network nodes, a covertly deployed WSN would be an excellent way to secretly monitor a hostile force or installation without need for maintenance or personnel. Such covert networks could be tied into a satellite data link, providing constant and instantaneous information to command centers anywhere on the globe [2].

3 Background

Wireless sensor networks have gained a wide range of attention in the past decade due to their promise to provide reliable, low maintenance, and relatively low cost sensors that can be quickly deployed into a wide variety of applications. Wireless sensor networks can generally be broken up into two categories: structured and ad-hoc. Structured WSNs consist of WSN networks with a planned deployment of each sensor node with regard to its location. Such deployments might be seen in industrial applications where Wireless Sensor Networks are replacing traditional wired sensors (such as safety valve monitoring at an oil refinery). Ad-hoc WSNs do not have planned deployments. The sensor nodes are distributed across an area of interest and are allowed to set up their own routing structure with respect to the base station(s). Ad-hoc WSNs typically consist of many more nodes than a structured WSN, as a higher density of nodes is required to ensure that fault tolerant wireless communication is possible between all nodes in the network and the base station. This project will focus on ad-hoc wireless sensor networks, as ad-hoc WSNs are more suited to military applications. An ad-hoc wireless sensor network is more conducive to surveillance over harsh terrain in remote geographic locations. An ad-hoc WSN gives the user the ability to deploy the network quickly; such networks could be quickly deployed by fast moving ground forces or military aircraft [1].

There are a number of different protocols and hardware sets that can be utilized for Wireless Sensor Networks. One of the first operating systems developed specifically for Wireless Sensor Networks is the 'Tiny' operating system, known as TinyOS. It was developed at the University of California at Berkeley beginning in 1999. TinyOS is a Linux based operating system that is significantly parsed down so that it can be utilized by resource limited WSN nodes. It is written in nesC, which is a variant of the C programming language and is 'event driven,' meaning it does not behave like many other operating systems when dealing with system processes. The entire operating system is only capable of performing one process at a time, and it does not provide a means to prioritize the order in which processes run. TinyOS utilizes many short programs, known as 'event handlers,' to handle large tasks that the node may be asked to perform, to include data routing [18].

4 Related Work

Security in Wireless Sensor Networks is an issue of critical importance to the development of WSN technology as a whole. A significant amount of research has been invested into solving some of the security issues that Wireless Sensor Networks face, to include intrusion detection, host authentication, and data sinkhole mitigation. In [11], authors I. Krontiris *et al.* propose a WSN implementation that would be able to detect sinkhole attacks in Wireless Sensor Networks that utilize the MintRoute protocol (a routing protocol that is similar to the Collection Tree Protocol). Such a system could enable a WSN to quickly detect an attack and trigger its defense mechanisms in order to reduce the volume of data. In [6], authors U. Colesanti and S. Santini have performed an in depth evaluation of the Collection Tree Protocol. Their research explains the inner workings of the CTP protocol in depth and tests the Collection Tree Protocol under several different conditions.

In [10], authors J. Deng *et al.* enumerate a "Intrusion-tolerant routing protocol for Wireless Sensor Networks: INSENS." The INSENS protocol aims to reduce the impact an adversary could have on a WSN by utilizing a number of security features to include light cryptography, positive host identification, and network analysis at the base station. In a similar line of research, T Shu *et al.*[8] proposes a method to defeat sinkhole attacks through the utilization of "randomized dispersive routes." Under this method, network messages are broken up into many 'shares' that are distributed throughout the network before they converge on the base station. Once a set number of shares successfully arrive at the base station, the data from the origin node can be reassembled. Neither of the routing schemes enumerated above utilize a purely routing based approach to mitigating the sinkhole problem in WSNs. Our approach uses only changes to network routing in order to avoid sinkholes.

5 Preliminary Project Work

To gain a better understanding of Wireless Sensor Networks, a portion of project time was spent working with actual WSN hardware. Two types of Wireless Sensor Network nodes were studied. The first was the MICA2 Wireless Measurement System. The MICA2 is commercially available through Crossbow Technology. The MICA2 platform is small, measuring only 2.25 x 1.25 x 1.0 inches with its sensor board and battery pack attached. The unit is powered by two standard AA batteries and has a battery life of up to one year

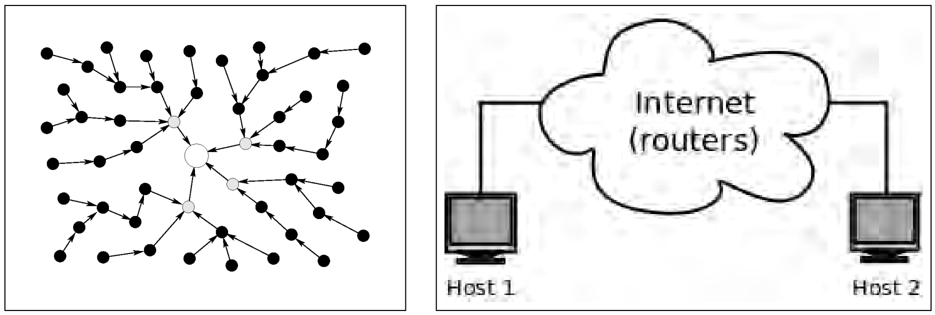
under continuous operation, given that it is calibrated to use a power saving 'sleep' mode that reduces the number of transmissions and computations that the node performs [17]. The MICA2 has only 128 KB of program memory and the processor only draws 8 mA of current while active and only 15 μA in sleep mode. The transmitter draws 27 mA of current when transmitting and 10 mA while in receive mode. It is important to note the high cost of radio transmission in terms of battery drain. The second WSN hardware platform used in project work was the IRIS Wireless Measurement System. The IRIS WSN is essentially an improvement on the MICA2 node, and is very similar in terms of size and capabilities [16]. Both WSN node platforms run the TinyOS operating system and are capable of implementing the Collection Tree Protocol (CTP). TinyOS applications were built, compiled, installed, and run on both the MICA2 and IRIS hardware platforms. Due to the limited memory and minimalist premises behind Wireless Sensor Networks, the installation of new programs on WSN nodes becomes a somewhat difficult task. In order to implement a new application on the MICA2 or IRIS platform, the entire operating system must be recompiled and then reloaded individually on to each sensor node. This process is very time consuming and would generally only be performed as part of a major operating system upgrade.

6 Security in Wireless Sensor Networks

Despite the immense potential of sensor networks, the low cost and small size of a single sensor node severely restricts an individual node's computing power and memory. On top of this, a single unit's lifetime is dictated by its least sophisticated technology – the battery. Any activity on the part of a node, in terms of computation and transmission, directly affects the lifetime of the entire unit. In addition, while the computing power of a comparable sized unit may increase following Moore's law[1], it is likely that more inexpensive productions of the same unit will be chosen over more capable components [3]. Thus, security in sensor networks must be designed and implemented with energy and computing efficiency in mind.

In addition to battery and computational limitations, wireless sensor networks face another unique problem – node capture. In typical network security schemes, it is assumed that the individual hosts (nodes) are safe from physical capture or tampering. In WSNs, nodes may need to be placed in hostile and or easily accessible locations, meaning that the system must

[1] Moore's law is the observation that the number of transistors that can be placed on a circuit doubles roughly every two years



(a) WSN many-to-one tree structured routing

(b) Internet host-to-host routing

Figure 1: Comparision of Internet routing and WSN routing

be designed so that the capture or destruction of a node will not disrupt the overall data transfer capability of the network [4]. Additionally, it can be assumed than an adversary will be able to extract critical data from a captured node. She may then be able to use this data to deny service to the network, or otherwise exploit the network's security system [5]. Threats on WSNs present a unique challenge in that many traditional computer network security solutions do not directly apply to a WSN.

On the Internet, hosts generally communicate in a one to one fashion, that is, one host uses the network to communicate with only one other specifically addressed host. Wireless sensor networks, on the other hand, do not communicate in a one to one fashion. Communication patterns in a WSN can be broken down into three basic types: many to one, one to many, and many to many [3]. The pattern of many to one communication stems from the layout of a typical wireless sensor network: a single 'base station' is responsible for collecting data from many different nodes. The base station is interested in aggregating data from a network composed of n nodes, where $n \gg 1$. The reverse is also true, resulting in a one to many communication pattern. This type of communication occurs when the base station wishes to send out configuration information to every node within the network, resulting in a multicast message that is generated at the base station of the network and is desseminated to every node in the network. Lastly, nodes within the network may want to exchange information with other nodes within the network (communication that excludes the base station). Such communication may occur when nodes are participating in the exchange of local routing information, aggregating data within a neighborhood of nodes, or 'voting' in an effort to detect an illegitimate node in the network [1]. For the purpose of this research, we will focus on WSN communication that is many to one.

The host to host nature of Internet communication generally follows a simple one to one communication model, where a host is capable of sending out messages with a foreign address and receiving data that has been addressed to it. The host is able to hand off a packet to a router that has a wider knowledge of the network topology. The router then sends the packet to a series of other routers that guide the message datagram to its eventual destination. Under this model, a host is only expected to know the address of the host that the message will be sent to. Conversely, in the role of a receiver, the host only has to listen for packets that are addressed to it. Routing across the network is not handled by hosts; it is handled instead by routers that are built and configured specifically for that task. In a wireless sensor network, routing is handled differently. Every node within a wireless sensor network can be expected to act as a receiving node, a transmitting node, and a routing node. Depending on the layout of the sensor network at deployment time, the topology of the network can leave any sensor in the network in one or a combination of all three of the roles mentioned above. This difference between Internet routing and WSN routing further complicates the application of Internet based networking protocols to WSN topologies.

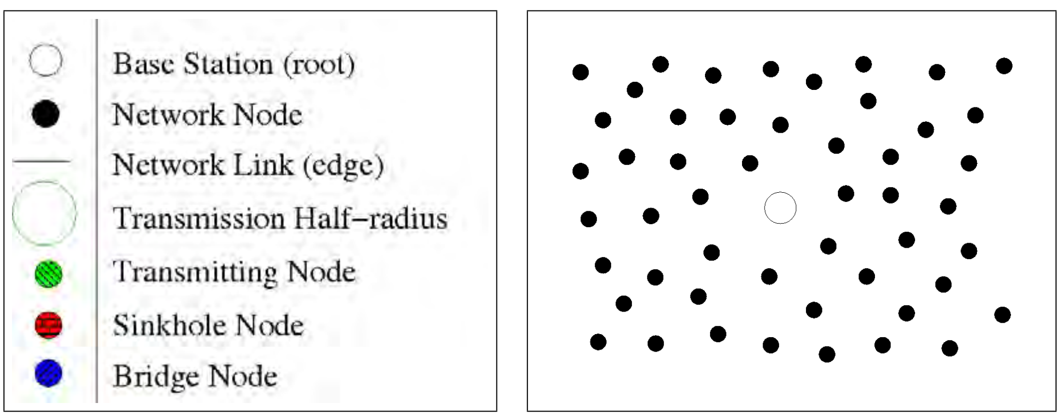
A majority of secure traffic over the Internet utilizes the Transmission Control Protocol. The Transmission Control Protocol utilizes two features that are generally not implemented in WSN protocols in order to save battery life (too many extra transmissions). The first feature is that of 'three way handshaking.' Before data is transmitted between two hosts on the Internet, TCP ensures that there is a valid and reliable connection between the two hosts. The sender initiates the 'handshake' by first querying the receiver. Upon receipt of the query, the receiving host will then send back an acknowledgement (known as an ACK), which lets the originating host know that a data exchange session has been set up between the sender and the receiver. Finally, the sending host sends its data to the receiving host (which also implicitly serves as an acknowledgement – the third part of the handshake). Three way handshaking allows a data transmission session to be set up between two hosts before actual information is exchanged. The second feature of TCP is an extension of the first – the use of acknowledgments. The Transmission Control Protocol performs accounting on the information that is transfered between hosts. Each datagram message that is exchanged between the two hosts is acknowledged in an ACK process that is similar to the three way handshake described above. If transmitted data is incomplete or lost, TCP will retransmit the data to ensure its accurate and complete arrival. Unfortunately, most Wireless Sensor Network protocols do not implement acknowledgements due to the extra transmissions that are required to ensure data delivery [15].

Internet	Wireless Sensor Networks
One to One communication	Many to One Communication
Assumption of host's physical security	No assumption of node security
Strong cryptography	Weak or no cryptography
Transmission Acknowledgements	Little or no acknowledgements

Table 1: Comparison of differences between Internet and WSN communication protocols

7 Wireless Sensor Network Topology

In order to understand the establishment of routing protocols in wireless sensor networks, we must first understand the properties of the network graph. We define the network graph to be the set of all WSN nodes, to include the base station (the base station is also commonly referred to as the 'root' or 'sink' of the network). We assume that the uninitialized network graph is a connected, undirected graph, meaning that every node in the network is adjacent to at least one other node in the network (they can communicate wirelessly). Please reference Figure 2 and Figure 3 for a further visual explanation.

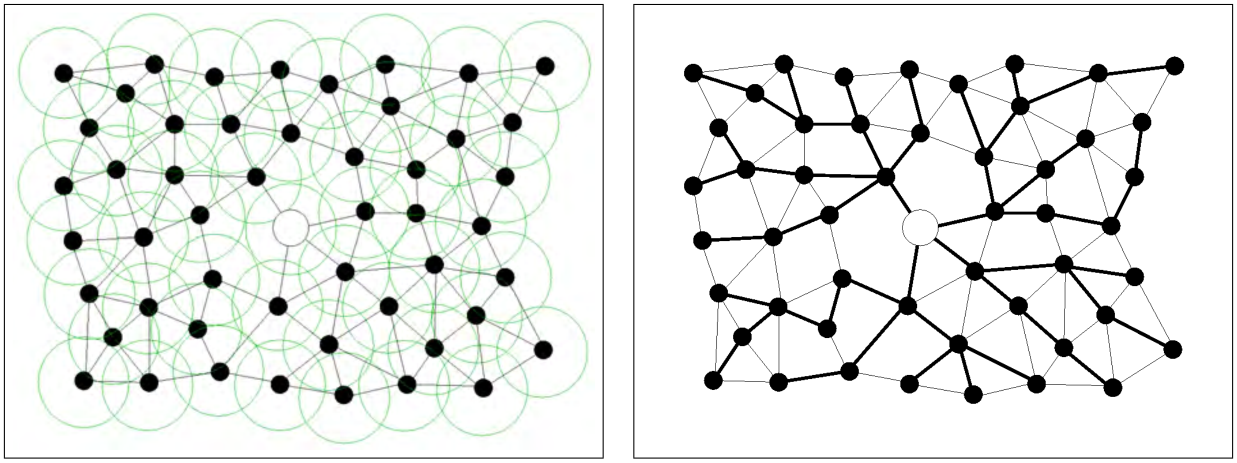


(a) Key for network diagrams

(b) An uninitialized WSN network with nodes and a base station

Figure 2

Adjacency in a wireless sensor network is dictated by the radius of communication R_c and is best described as a *unit disk graph*: each node lies at the center of a unit disk (with unit radius r), and an edge is defined whenever two disks overlap. For sensor networks, we then have $r = R_c/2$ [9]. This requirement on the edges imposes a sense of distance in the graph. It is important to note that arbitrary edge sets are not possible; in particular, edges that



(a) Unit disk graph with radii 'r.' Overlapping 'r' indicates a possible network edge

(b) All possible connections of the network graph, with higher quality network connections represented in bold

Figure 3: An arbitrary network graph with possible network connection edges given the 'r' of each node

exceed $2r$ in length cannot occur, so widely distributed edges cannot be connected directly, and hence must be connected by a path through the graph as shown in Figure 3.

The tree structured routing utilized by wireless sensor networks is established through the utilization of some sort of path metric. In general, the selection of edges is based on the 'best path' between two nodes, given the path metric. The Collection Tree Protocol (CTP) seeks to find the best path to the root node by transmitting as few times as possible (it seeks to transmit over the most reliable path to the base station). This path metric in CTP is known as the ETX value which stands for 'Expected Transmissions'). The ETX value represents the predicted number of node transmissions that will need to occur for a message to reach the base station. The ETX value is established during network setup and is based on the number of transmissions that are necessary to successfully transmit a routing query message between two nodes. An optimal ETX value between any set of nodes is '1,' as only one transmission is required to successfully transmit the data. The base station is a special case, as it does not need to transmit in order to communicate with itself, thus, the base station has an ETX value of '0.' Every other node in the network will have an ETX value that is greater than '1,' and the ETX value of any node in the network will the be the sum of the ETX value of its parent and the ETX value of the link between the aforementioned node and its parent. A valid data transmission over a routing tree using CTP is shown in Figure 4

We model a sensor network as a unit disk graph $G(V, E)$, composed of the set $V, |V| = N$ of vertices (sensors) and the set E of edges determined by radio reception.

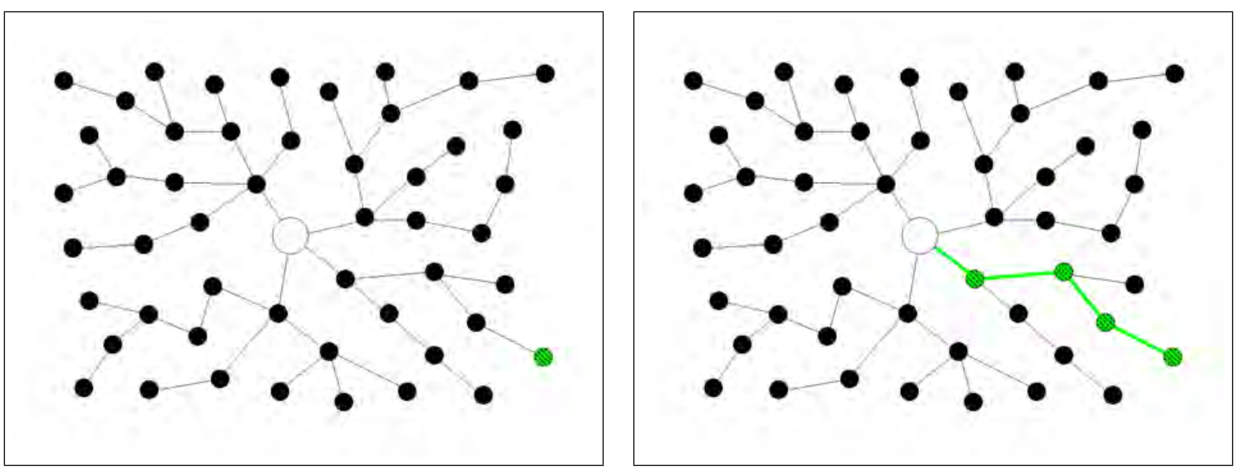
The tree $T(V, E') \subseteq G(V, E)$ is defined by the TinyOS Collection Tree Protocol, rooted at the base station. We will usually denote this tree simply as T_{CTP}.

We make the following assumptions about the sensor network:

- Each sensor node is identical in terms of initial battery life, transmitter and receiver capacity.
- Only one base station is placed.
- Once placed, the sensors are fixed.
- The sensors are uniformly distributed; for any sensor v and neighborhood

  $$N_v \triangleq \{v_i \mid d_r(v, v_i) < r_{\text{radio}}\}$$

  we have, on average, $|N_v| = k$ for any $v \in G$, with sufficient small variance to allow for simple analysis.



(a) The green node is attempting to transmit data to the base station

(b) The data follows the path directed by the network tree and successfully reaches the base station

Figure 4: Normal data routing in a WSN

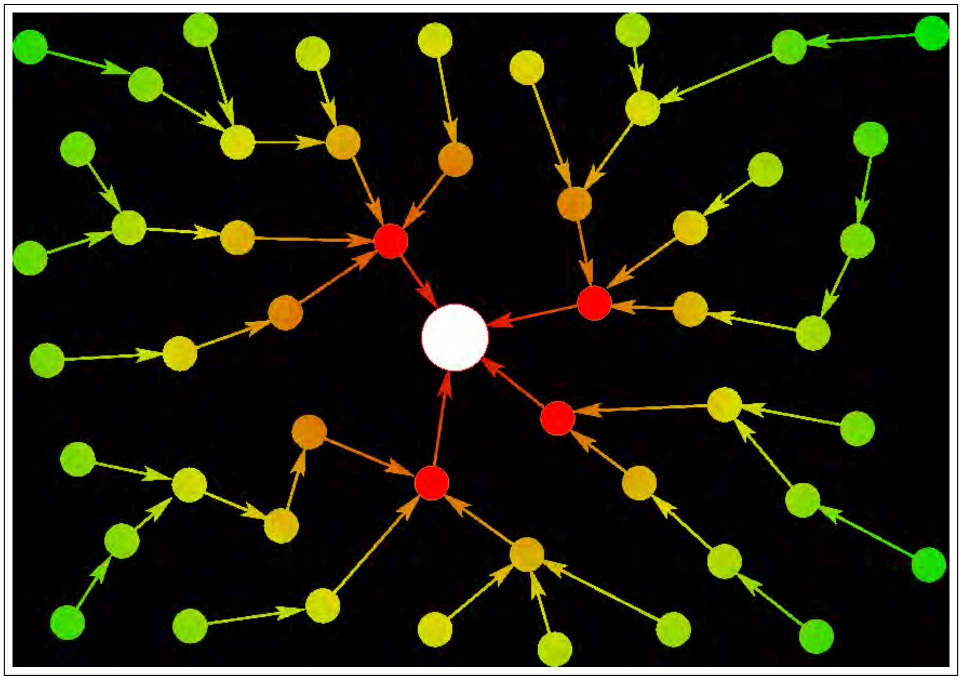


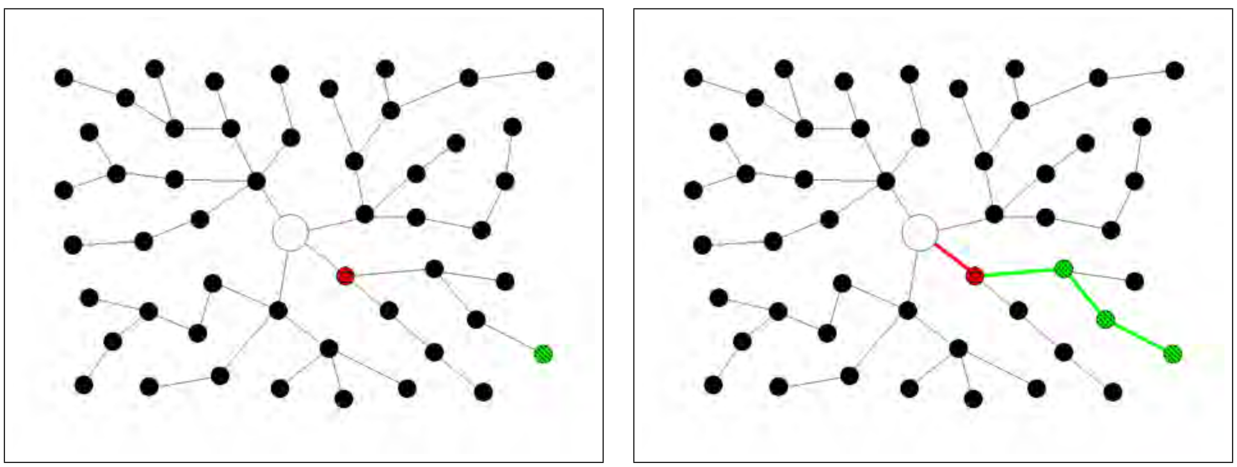
Figure 5: A depiction of the risk gradient, where nodes in green represent a low risk of data loss,and nodes in red represent a high risk of data loss

8 Threat Model

The essential function of a sensor network is to report data. The data flows across the deployment region, routed via the minimum weight spanning tree (MST). Each sensor routes data along the minimum weight path to a base station that acts as a root node (we will use 'node' and 'sensor' interchangeably when there is no confusion). For the purpose of our research, we assume that the base station is secure and has not been tampered with by an adversary. This is a reasonable assumption, as the compromise of the base station would allow the adversary to control the entire WSN and serves to trivialize the avoidance of a routing sinkhole, which is our project focus.

A sensor node that has been compromised by an attacker can act as a 'sinkhole' and manipulate all network data that is forwarded to it. Our research focuses on an adversary that controls a sinkhole in the network and chooses to drop all network data that is forwarded to it. In particular, the adversary influences the routing so that it maximizes the amount of traffic flowing to it. This is the 'sinkhole' attack and is the focus of this work. We assume that the adversary is able to completely compromise one sensor. The adversary will then

have the ability to influence how the minimum spanning tree is established. A compromised sensor would be able to advertise a favorable metric so as to be included in the tree as a routing node instead of a leaf node. We assume that the adversary may enhance the sensor to support this metric. In this way, the compromised sensor will receive traffic from downstream nodes for examination (whereas a leaf node does not have downstream nodes). The adversary may achieve her goal through a combination of positioning herself close to the base, or influencing how the minimum spanning tree is established through fraudulent route costs, wherein a compromised sensor will advertise a favorable metric so as to be included in the tree as a routing node instead of a leaf node. We assume that the adversary only drops traffic. This means that the adversary will try to influence as much traffic as possible by positioning herself close to the base, resulting in a "risk gradient" where nodes that are far from the base are less likely to be compromised, as shown in Figure 5.



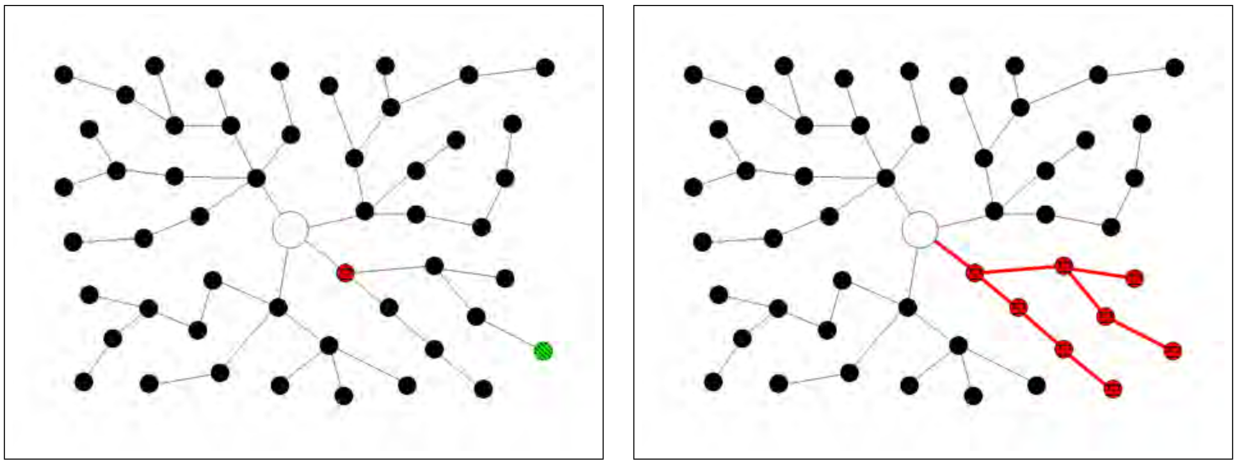
(a) The green node is attempting to transmit data to the base station in the presence of a sinkhole

(b) The data follows the path directed by the network tree and is intercepted by the sinkhole before it reaches the base station

Figure 6: WSN Routing Example

The position of a sinkhole node in the network will affect the impact that the sinkhole is able to have on the network as a whole. In a network that utilizes tree structured routing, every node in the network must rely on its parent to forward data closer to the base station. In a sense, tree structured routing creates a communication chain that can be broken by removing a single link. The closer the broken link is to the base station, the more links that will be severed from the base station. This idea ties into the risk gradient depicted in Figure 5. If a sinkhole node happens to be positioned at the root of a subtree, then the

sinkhole will be able to disconnect that entire subtree from the base station. Depending on the distribution of nodes in the sensor network, this could mean that a large geographic area of the sensor network will be unable to report its data.



(a) A WSN with a sinkhole, given the green transmitting node and the red sinkhole

(b) The sinkhole is at the root of the subtree, which effectively cuts off the rest of that subtree from the network

Figure 7: The effect of a sinkhole in a WSN

9 Description of Protocols

9.1 Generic Protocol Description

In order to better describe the complex distributed protocols enumerated in this project, we will first define an abstract distributed protocol. Our abstract scenario will describe barking dogs in a neighborhood. Our distributed protocol will give instructions for a dog to execute once an event occurs. Each dog in the neighborhood executes the same protocol.

When (I hear another dog bark)
Then
  1 I bark once

This abstract protocol can be used to imagine how a WSN initializes the routing infrastructure of the network. A dog that is in the center of his neighborhood would initiate a bark. This bark would then move outward towards the edges of the neighborhood until every dog

in the neighborhood is barking. The CTP protocol initiates itself in a similar manner, except that beacons are used instead of barks, and that many of the events are triggered by timers in order to reduce the number of transmissions required by each node.

9.2 The Collection Tree Protocol

The Collection Tree Protocol (CTP) is the standard protocol that our project aims to improve upon. CTP attempts to transmit data over the lowest cost path. In simple terms, the 'cost' of a spanning tree is the sum over all edges of the transmission quality. This quality is captured by the Expected Transmissions value (ETX), and is used by CTP for the construction of the minimal spanning tree. From the description: "CTP is a tree-based collection protocol. Some number of nodes in a network advertise themselves as tree roots. Nodes form a set of routing trees to these roots. CTP is address-free in that a node does not send a packet to a particular root; instead, it implicitly chooses a root by choosing a next hop. Nodes generate routes to roots using a routing gradient." [7] We seek to expand CTP so that it supports bridge discovery. The challenge is to implement our changes to the source code in such a way that does not break the protocol or significantly add to the complexity of the protocol in terms of battery cost (extra transmissions and computations). We are most concerned with taking advantage of the distributed nature of the protocol so as to keep the routing complexity on an individual node to a minimum.

Our modification of CTP will occur inside the routing engine. We will focus on four functions within the code that have important roles in initializing routing within CTP. We refer to these functions as *event handlers*, as they are executed when a specific routing related event occurs. These event handlers allow CTP to build and update the routing tables[2] necessary for the protocol to successfully implement tree structured routing. The four event handlers we will modify are: the Send Beacon Handler, the Beacon Message Received Handler, the Table Update Handler, and the Update Route Handler. We also describe the Beacon Timer Handler, as it directly influences two of the other event handlers that we modify within the routing engine.

[2] CTP stores the information of neighboring nodes in a data structure know as a *Routing Table*. By default, this data structure is limited to 10 entries.

Beacon Timer Handler

When	(The send beacon timer expires)
Then	**1** Reset send beacon timer **2** Send event to *Send Beacon Handler* **3** Send event to *Update Route Handler*

Source code[3]:

```
void CtpRoutingEngine::event_BeaconTimer_fired(){
    if(radioOn && running){
        if(!tHasPassed){
            post_updateRouteTask();
            post_sendBeaconTask();
            trace()<<"Beacon timer fired.";
            remainingInterval();
        }
        else{
            decayInterval();
        }
    }
}
```

Send Beacon Handler

When	(A send beacon event is received)
Then	**1** Send a beacon with my information: ID, Parent ID, ETX

Source code:

```
void CtpRoutingEngine::sendBeaconTask(){
        error_t eval;
        if(sending){
                return;
        }

        beaconMsg->setOptions(0) ;

        if(cfe->command_CtpCongestion_isCongested()){
                beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_ECN);
        }

        beaconMsg->setParent(routeInfo.parent);
        if(state_is_root){
                beaconMsg->setEtx(routeInfo.etx);
        }
```

[3] A complete listing of all source code is printed in the appendix

```
        else if(routeInfo.parent == INVALID_ADDR){
                beaconMsg->setEtx(routeInfo.etx);
                beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_PULL);
        } else{
                beaconMsg->setEtx(routeInfo.etx + le->command_LinkEstimator_getLinkQuality(
                    routeInfo.parent));
        }

        trace()<<"sendBeaconTask - parent: "<<(int)beaconMsg->getParent()
    <<" etx: "<<(int)beaconMsg->getEtx();

        beaconMsg->getRoutingInteractionControl().lastHop = self ; // ok
        eval = le->command_Send_send(AM_BROADCAST_ADDR, beaconMsg->dup());

        if(eval == SUCCESS){
                //statistics
                collectOutput("Ctp_Beacons","Tx") ;
                sending = true;
        } else if(eval == EOFF){
                radioOn = false;
                trace()<<"sendBeaconTask - running: "<<running<<" radioOn: "<<radioOn;
        }
}
```

Beacon Message Received Handler

When (A beacon message is received)

Then

1 Read neighbor beacon information: ID, Parent ID, ETX

2 Check parent ID

3 If parent ID matches my own ID, then stop

4 Calculate message ETX using the link estimator

5 Calculate new ETX by adding neighbor ETX to message ETX

6 Send event to *Table Update Handler*

Source code:

```
void CtpRoutingEngine::event_BeaconReceive_receive(cPacket* msg){
    Enter_Method("event_BeaconReceive_receive");
    am_addr_t from;
    bool congested;

    //statistics
    collectOutput("Ctp_Beacons","Rx");

    from = command_AMPacket_source(msg);
    CtpBeacon* rcvBeacon = check_and_cast<CtpBeacon*>(msg);
    congested = command_CtpRoutingPacket_getOption(msg,CTP_OPT_ECN);
```

```
    trace()<<"BeaconReceive.receive␣-␣from␣"<<(int)from<<"␣[parent:␣"
    <<(int)rcvBeacon->getParent()<<"␣etx:␣"<<(int)rcvBeacon->getEtx()
    <<"]";

    //update neighbor table
    if(rcvBeacon->getParent() != INVALID_ADDR){
        // If this node is a root, request a forced insert in the link
        // estimator table and pin the node.
        if(rcvBeacon->getEtx() == 0){
            trace()<<"from␣a␣root,␣inserting␣if␣not␣in␣table␣"
            <<"my_ll_addr:␣"<<my_ll_addr;
            le->command_LinkEstimator_insertNeighbor(from);
            le->command_LinkEstimator_pinNeighbor(from);
        }
        routingTableUpdateEntry(
                from,rcvBeacon->getParent(),
                rcvBeacon->getEtx();

        command_CtpInfo_setNeighborCongested(from,congested) ;
    }

    if(command_CtpRoutingPacket_getOption(msg, CTP_OPT_PULL))
        resetInterval();

    delete msg ;
}
```

Table Update Handler

When	(A table update event is received)
Then	**1** Read new neighbor information: ID, Parent ID, ETX **2** If routing table is full, then stop **3** Add new neighbor information to the routing table

Source code:

```
error_t CtpRoutingEngine::routingTableUpdateEntry(
    am_addr_t from,
    am_addr_t parent,
    uint16_t etx{

    uint8_t idx;
    uint16_t  linkEtx;
    linkEtx = evaluateEtx(le->command_LinkEstimator_getLinkQuality(from));

    idx = routingTableFind(from);
    if(idx == routingTableSize){
        trace()<<"routingTableUpdateEntry␣-␣FAIL,␣table␣full";
        return FAIL;
    }
```

```
    else if(idx == routingTableActive){
        if(passLinkEtxThreshold(linkEtx)){
            routingTable[idx].neighbor = from;
            routingTable[idx].info.parent = parent;
            routingTable[idx].info.etx = etx;
            routingTable[idx].info.haveHeard = 1;
            routingTable[idx].info.congested = false;
            routingTableActive++;
            trace()<<"routingTableUpdateEntry - OK, new entry";
        }
        else
            trace()<<"routingTableUpdateEntry - Fail, link quality("
            <<(int)linkEtx<<") below threshold";

    }
    else{
        //found, just update
        routingTable[idx].neighbor = from;
        routingTable[idx].info.parent = parent;
        routingTable[idx].info.etx = etx;
        routingTable[idx].info.haveHeard = 1;
        trace()<<"routingTableUpdateEntry - OK, updated entry";
    }
    return SUCCESS;
}
```

Update Route Handler:

When	(An update route event is received)
Then	**1** Read all neighbor information from the table: ID, Parent ID, ETX **2** Pick the neighbor with the best ETX **3** Set that neighbor as the new parent

Source code:

```
void CtpRoutingEngine::updateRouteTask(){

    uint8_t i;
    routing_table_entry* entry;
    routing_table_entry* best;
    uint16_t minEtx;
    uint16_t currentEtx;
    uint16_t linkEtx, pathEtx;

    if (state_is_root)
        return;

    best = NULL;
    /* Minimum etx found among neighbors, initially infinity */
    minEtx = MAX_METRIC;
```

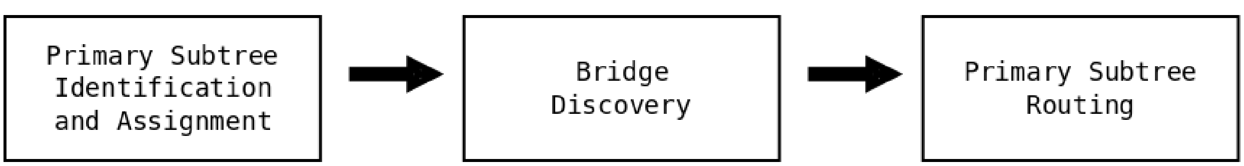
```
/* Metric through current parent, initially infinity */
currentEtx = MAX_METRIC;

trace()<<"updateRouteTask";

/* Find best path in table, other than our current */
for(i = 0; i < routingTableActive; i++){
    entry = &routingTable[i];

    // Avoid bad entries and 1-hop loops
    if(entry->info.parent == INVALID_ADDR ||
       entry->info.parent == my_ll_addr){

        trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "
        <<(int)entry->neighbor<<" parent: "<<entry->info.parent
        <<"  etx: NO_ROUTE]";

        continue;
    }

    // Compute this neighbor's path metric
    linkEtx = evaluateEtx(le->command_LinkEstimator_getLinkQuality(entry->neighbor));

    trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "<<(int)entry->neighbor
    <<" parent: "<<entry->info.parent<<"  etx: "<<(int)linkEtx
    <<"]";

    pathEtx = linkEtx + entry->info.etx;
    /* Operations specific to the current parent */
    if(entry->neighbor == routeInfo.parent){
        trace()<<"already parent";

        currentEtx = pathEtx;

        /* update routeInfo with parent's current info */
        routeInfo.etx = entry->info.etx;
        routeInfo.congested = entry->info.congested;
        continue;
    }

    /* Ignore links that are congested */
    if(entry->info.congested)
        continue;
    /* Ignore links that are bad */
    if(!passLinkEtxThreshold(linkEtx)){
        trace()<<"did not pass threshold.";
        continue;
    }

    if(pathEtx < minEtx){
        minEtx = pathEtx;
        best = entry;
```

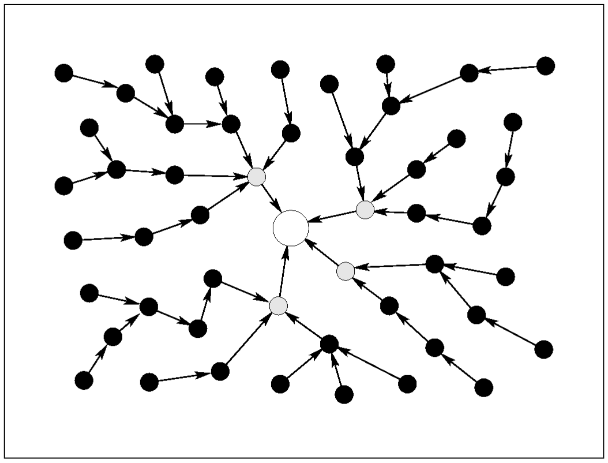
```
        }
    }
    if(minEtx != MAX_METRIC){
        if(currentEtx == MAX_METRIC ||
           (routeInfo.congested && (minEtx < (routeInfo.etx + 10))) ||
           minEtx + PARENT_SWITCH_THRESHOLD < currentEtx){

            parentChanges++;

            trace()<<"Changed parent. from "<<(int)routeInfo.parent<<" to "
            <<(int)best->neighbor;

            le->command_LinkEstimator_unpinNeighbor(routeInfo.parent) ;
            le->command_LinkEstimator_pinNeighbor(best->neighbor) ;
            le->command_LinkEstimator_clearDLQ(best->neighbor) ;

            routeInfo.parent = best->neighbor;
            routeInfo.etx = best->info.etx;
            routeInfo.congested = best->info.congested;
        }
    }
    else if(!justEvicted &&
            currentEtx == MAX_METRIC &&
            minEtx != MAX_METRIC)
        signal_Routing_routeFound();

    justEvicted = false;
}
```

9.3 Protocol Modification

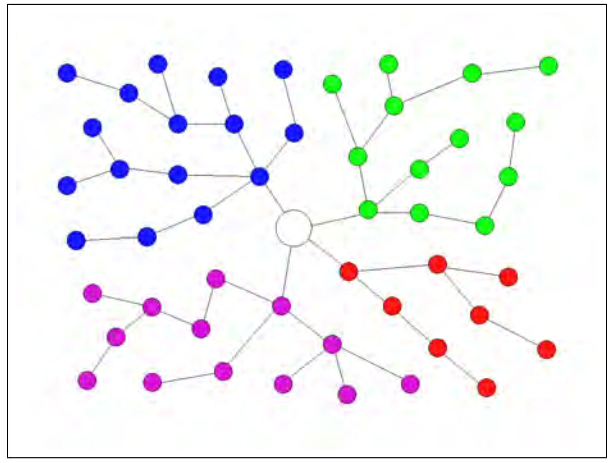
The Collection Tree Protocol (CTP), which is described in detail above, is used as the basis for our routing protocol. Our work has added the awareness of principle subtrees to CTP. Our changes to CTP are described below.

Our protocol:





(a) Directed tree utilized by the network protocol for data forwarding



(b) The different subtrees of the entire network tree are highlight in different colors

Figure 8: Network tree topology that illustrates the multiple principle subtrees of the entire network

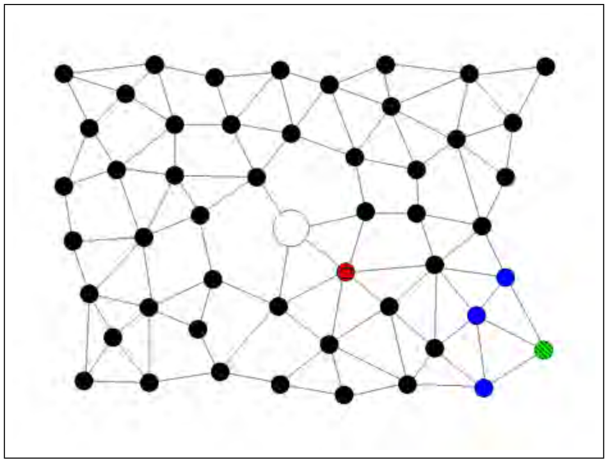
Principle Subtree Identification and Assignment

- After CTP network setup is complete, determine which nodes have the base station as a parent
- Identify these nodes as the roots of the principle subtrees
- Each of these root nodes receives an index value assigned by the base station
- The subtree roots then disseminate their principle subtree index (PST ID) to the members of their principle subtree

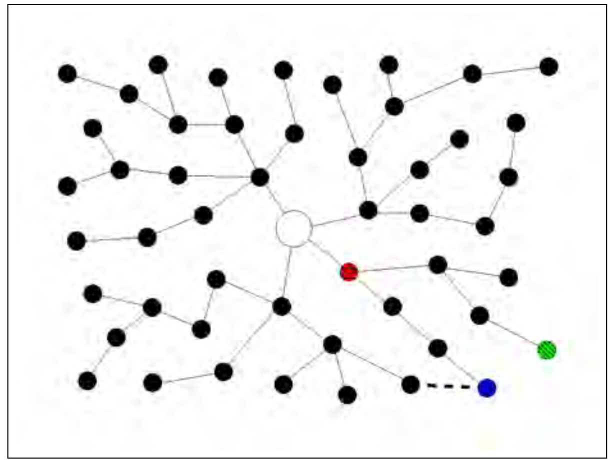
Bridge Discovery - reference Figure 9

- Each node in the network queries its list of neighbors and obtains their PST ID
- If a node has a neighbor with a PST ID value different from its own, then the neighbor node is a bridge
- Each bridge in a principle subtree T_j, with index j, that contains the sinkhole 're-runs' the CTP tree establishment protocol by advertising a very low-cost route to the base. This advertisement is limited to only those sensor nodes in T_j, and results in multiple CTP tree roots with routes to the base. Sensor node routing tables are modified to keep track of whether next hops (to parents) are routed towards bridges, giving nodes the

chance to pick a next hop at random from among the available bridges. This approach is feasible since we have implemented the ability for sensor nodes to determine their subtree index as part of the bridge discovery simulations.



(a) The transmitting node has multiple neighbors that it can potentially forward data to, represented by the blue nodes

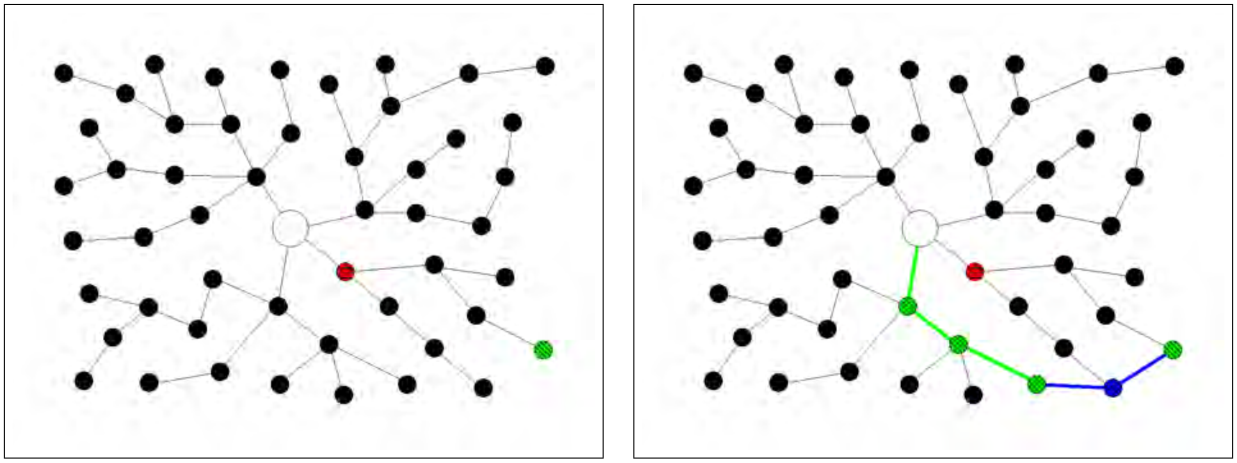


(b) One neighbor of the transmitting node is a bridge to another subtree, which is indicated by the blue node and its dashed line to the alternate subtree

Figure 9: Bridge node identification

Principle Subtree Routing - reference Figure 10

- If a node suspects that there is a sinkhole in its subtree, it chooses a bridge node and routes in that direction (adding the bridge node ID to the packet to ensure proper downstream routing)
- A bridge node that receives data from a different subtree will forward the data to the base station via the Collection Tree Protocol



(a) The transmitting node is attempting to transmit data to the base station in the presence of a sinkhole

(b) The transmitting node routes data to a bridge node, who then routes the data to a different subtree. Once in a different subtree, the data is routed to the base station via the normal directed network tree

Figure 10: Principle Subtree Routing using a bridge between principle subtrees

Our changes to the CTP protocol occur within four event handlers that are a part of the CTP routing engine: the Send Beacon Handler, the Beacon Message Received Handler, the Table Update Handler, and the Update Route Handler. We modify these functions so as to implement bridge descovery through the addition of a priniciple subtree index (PST_ID). Inclusion of the PST_ID in the protocol will only add two bytes[4] to each beacon message.

Beacon Timer Handler[5]

When	(The send beacon timer expires)
Then	**1** Reset send beacon timer **2** Send event to *Send Beacon Handler* **3** Send event to *Update Route Handler*

[4]One byte is equal to 8 bits.

[5]We do not modify this event handler; however, it is important to understand it's function as it is responsible for triggering the *Send Beacon Handler* and the *Update Route Handler*.

Send Beacon Handler[6]

When	(A send beacon event is received)
Then	
	1 Send a beacon with my information: ID, Parent ID, ETX, *PST_ID*

Source code:

```
void CtpRoutingEngine::sendBeaconTask() {
        error_t eval;
        if(sending){
                return;
        }

        beaconMsg->setOptions(0);

        if(cfe->command_CtpCongestion_isCongested()){
                beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_ECN);
        }

        beaconMsg->setParent(routeInfo.parent);
        if(state_is_root){
                beaconMsg->setEtx(routeInfo.etx);
        }
        else if(routeInfo.parent == INVALID_ADDR){
                beaconMsg->setEtx(routeInfo.etx);
                beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_PULL);
        } else{
                beaconMsg->setEtx(routeInfo.etx + le->command_LinkEstimator_getLinkQuality(
                    routeInfo.parent));
        }

        beaconMsg->setPstId(my_pstId);

        trace()<<"sendBeaconTask - parent: "<<(int)beaconMsg->getParent()
    <<" etx: "<<(int)beaconMsg->getEtx();

        beaconMsg->getRoutingInteractionControl().lastHop = self ; // ok
        eval = le->command_Send_send(AM_BROADCAST_ADDR, beaconMsg->dup());

        if(eval == SUCCESS) {
                //statistics
                collectOutput("Ctp Beacons","Tx") ;
                sending = true;
        } else if(eval == EOFF) {
                radioOn = false;
                trace()<<"sendBeaconTask - running: "<<running<<" radioOn: "<<radioOn;
        }
}
```

[6]Changes to the original CTP protocol are printed in *italics*

Beacon Message Received Handler

When (A beacon message is received)

Then

1 Read neighbor beacon information: ID, Parent ID, ETX,*PST_ID*

2 Check parent ID

3 If parent ID matches my own ID, then stop

4 *If parent ID is root, set my PST_ID to my ID*

5 *If parent is not root, set my PST_ID to my parent's PST_ID*

6 Calculate message ETX using the link estimator

7 Calculate new ETX by adding neighbor ETX to message ETX

8 Send event to *Table Update Handler*

Source code:

```
void CtpRoutingEngine::event_BeaconReceive_receive(cPacket* msg){
    Enter_Method("event_BeaconReceive_receive");
    am_addr_t from;
    bool congested;

    //statistics
    collectOutput("Ctp_Beacons","Rx");

    from = command_AMPacket_source(msg);
    CtpBeacon* rcvBeacon = check_and_cast<CtpBeacon*>(msg);
    congested = command_CtpRoutingPacket_getOption(msg,CTP_OPT_ECN);

    trace()<<"BeaconReceive.receive - from "<<(int)from<<" [parent: "
    <<(int)rcvBeacon->getParent()<<" etx: "<<(int)rcvBeacon->getEtx()
   <<" pstId: "<<(int)rcvBeacon->getPstId()<<"]";

    //update neighbor table
    if(rcvBeacon->getParent() != INVALID_ADDR){
        // If this node is a root, request a forced insert in the link
        // estimator table and pin the node.
        if(rcvBeacon->getEtx() == 0){
            trace()<<"from a root, inserting if not in table "
           <<"my_ll_addr: "<<my_ll_addr;
            le->command_LinkEstimator_insertNeighbor(from);
            le->command_LinkEstimator_pinNeighbor(from);
            // since i hear root, i'm the root of a principle subtree
            my_pstId = my_ll_addr;
        }

        if(rcvBeacon->getEtx() == 0)
            routingTableUpdateEntry(
                from,rcvBeacon->getParent(),
                rcvBeacon->getEtx(),
```

```
                my_pstId);
        else
            routingTableUpdateEntry(
                from, rcvBeacon->getParent(),
                rcvBeacon->getEtx(),
                rcvBeacon->getPstId());

        command_CtpInfo_setNeighborCongested(from, congested) ;
    }

    if(command_CtpRoutingPacket_getOption(msg, CTP_OPT_PULL))
        resetInterval();

    delete msg ;
}
```

Table Update Handler

When	(A table update event is received)
Then	**1** Read new neighbor information: ID, Parent ID, ETX, *PST_ID* **2** If routing table is full, then stop **3** *If neighbor PST_ID is not equal to my PST_ID, I am a bridge* **4** Add new neighbor information to the routing table

Source code:

```
error_t CtpRoutingEngine::routingTableUpdateEntry(
    am_addr_t from,
    am_addr_t parent,
    uint16_t etx,
    am_addr_t pstId){

    uint8_t idx;
    uint16_t  linkEtx;
    linkEtx = evaluateEtx(le->command_LinkEstimator_getLinkQuality(from));

    idx = routingTableFind(from);
    if(idx == routingTableSize){
        trace()<<"routingTableUpdateEntry - FAIL, table full";
        return FAIL;
    }
    else if(idx == routingTableActive){
        if(passLinkEtxThreshold(linkEtx)){
            routingTable[idx].neighbor = from;
            routingTable[idx].pstId = pstId;
            routingTable[idx].info.parent = parent;
            routingTable[idx].info.etx = etx;
            routingTable[idx].info.haveHeard = 1;
            routingTable[idx].info.congested = false;
```

```
            routingTableActive++;
            trace()<<"routingTableUpdateEntry - OK, new entry";
            if (pstId != my_pstId)
                trace()<<"## my_pstId: "<<(int)my_pstId<<" neighbor pstId: "
                <<pstId;
        }
        else
            trace()<<"routingTableUpdateEntry - Fail, link quality ("
            <<(int)linkEtx<<") below threshold";

    }
    else{
        //found, just update
        routingTable[idx].neighbor = from;
        routingTable[idx].info.parent = parent;
        routingTable[idx].info.etx = etx;
        routingTable[idx].info.haveHeard = 1;
        trace()<<"routingTableUpdateEntry - OK, updated entry";
    }
    return SUCCESS;
}
```

Update Route Handler:

When	(An update route event is received)
Then	**1** Read all neighbor information from the table: ID, Parent ID, ETX, *PST_ID* **2** Pick the neighbor with the best ETX **3** Set that neighbor as the new parent

Source code:

```
void CtpRoutingEngine::updateRouteTask(){

    uint8_t i;
    routing_table_entry* entry;
    routing_table_entry* best;
    uint16_t minEtx;
    uint16_t currentEtx;
    uint16_t linkEtx, pathEtx;

    if (state_is_root)
        return;

    best = NULL;
    /* Minimum etx found among neighbors, initially infinity */
    minEtx = MAX_METRIC;
    /* Metric through current parent, initially infinity */
    currentEtx = MAX_METRIC;

    trace()<<"updateRouteTask";
```

```
/* Find best path in table, other than our current */
for(i = 0; i < routingTableActive; i++){
    entry = &routingTable[i];

    // Avoid bad entries and 1-hop loops
    if(entry->info.parent == INVALID_ADDR ||
       entry->info.parent == my_ll_addr){

        trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "
        <<(int)entry->neighbor<<" parent: "<<entry->info.parent
        <<"  etx: NO_ROUTE]";

        continue;
    }

    // Compute this neighbor's path metric
    linkEtx = evaluateEtx(le->command_LinkEstimator_getLinkQuality(entry->neighbor));

    trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "<<(int)entry->neighbor
    <<" parent: "<<entry->info.parent<<"  etx: "<<(int)linkEtx
    <<" pstId: "<<(int)entry->pstId<<"]";

    pathEtx = linkEtx + entry->info.etx;
    /* Operations specific to the current parent */
    if(entry->neighbor == routeInfo.parent){
        trace()<<"already parent";

        currentEtx = pathEtx;

        /* update routeInfo with parent's current info */
        routeInfo.etx = entry->info.etx;
        routeInfo.congested = entry->info.congested;
        continue;
    }

    /* Ignore links that are congested */
    if(entry->info.congested)
        continue;
    /* Ignore links that are bad */
    if(!passLinkEtxThreshold(linkEtx)){
        trace()<<"did not pass threshold.";
        continue;
    }

    if(pathEtx < minEtx){
        minEtx = pathEtx;
        best = entry;
    }
}
if(minEtx != MAX_METRIC){
    if(currentEtx == MAX_METRIC ||
```

```
            (routeInfo.congested && (minEtx < (routeInfo.etx + 10))) ||
            minEtx + PARENT_SWITCH_THRESHOLD < currentEtx){

             parentChanges++;

             trace()<<"Changed parent. from "<<(int)routeInfo.parent<<" to "
             <<(int)best->neighbor<<" with pstId "<<(int)best->pstId;

             le->command_LinkEstimator_unpinNeighbor(routeInfo.parent) ;
             le->command_LinkEstimator_pinNeighbor(best->neighbor) ;
             le->command_LinkEstimator_clearDLQ(best->neighbor) ;

             routeInfo.parent = best->neighbor;
             routeInfo.etx = best->info.etx;
             routeInfo.congested = best->info.congested;
             my_pstId = best->pstId;
        }
    }
    else if(!justEvicted &&
             currentEtx == MAX_METRIC &&
             minEtx != MAX_METRIC)
        signal_Routing_routeFound();

    justEvicted = false;
}
```

10 Simulations

Our research has utilized simulations in order to increase our understanding of how the CTP works, and also to show some principles of the network. Our network simulations utilize the Castalia v3.0 wireless sensor networks simulator, which operates on top of the OMNeT++ v4.1 network simulator. In the Castalia simulator, we have modified an implementation of the CTP protocol adapted for the C++ language used by Castalia. The original C++ code for the CTP protocol has been provided to us by the authors of [6].

In our project, we simulated networks of 50, 150, and 300 MICA2 WSN nodes with a single base station. The simulated WSN networks were programmed to use the CTP protocol for routing. After successfully simulating a network of 50 nodes, we moved on to simulating larger networks. In addition, we implemented a sinkhole node in the simulation network. The sinkhole actively sends and receives network beacons but does not forward actual application (sensor) data to the base station.

Data from sixty simulation runs are shown in Appendix A.1. The graph shows the effect a sinkhole can have on a WSN that utilizes CTP. The graph shows simulations on a 300 node network. In these simulation runs, thirty individual network topologies were used. Each topology was generated using a random uniform distribution. Within each distribution, two simulations were run: one that tested the network with a sinkhole, and one that tested the network without a sinkhole. The graph shows the individual topologies as two columns: the red column represents packet loss[7] (as a percentage) with a sinkhole, while the blue column represents the packet loss without a sinkhole. It is important to note that there can be packet loss when there is no sinkhole in the network – these losses can be attributed to simulated data collisions[8] or errors.

Simulations were also conducted using our modified protocol. Through the trace file generated during each simulation run, we were able to determine that bridge nodes were successfully discovered by our protocol.

11 Conclusion

This project has developed a novel strategy to mitigate sinkhole attacks in WSNs that utilize tree structured routing. Our work has shown that an implementation of this strategy in the CTP protocol is feasible based on simulations that show the existence of 'bridge nodes' in ad-hoc WSNs that are initialized using CTP. Future work on this area might include implementing our sinkhole mitigation strategy into a working version of the Collection Tree Protocol (CTP) and testing its effectiveness with respect to the overall data delivery ratio in the Castalia wireless sensor networks simulator.

References

[1] J. Yick, B. Mukherjee and D. Ghosal. Wireless Sensor Network Survey. *Department of Computer Science, University of California at Davis*, 2008.

[7] Packet loss refers to the percentage of data that was transmitted by a node in the network but was not successfully received by the base station.

[8] Data collisions occur when a node receives two or more data packets simultaneously and the node is forced to drop some or all of the data.

[2] M. Saraogi. Security in Wireless Sensor Networks. *Department of Computer Science, University of Tennessee, Knoxville*, 2005.

[3] C. Karlof and D. Wagner. Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures. *University of California, Berkeley.*

[4] A. Perrig, J. Stankovic, and D. Wagner. Security in Wireless Sensor Networks. In *Communications of the ACM* Vol 47 No 6, 2004.

[5] A. Vaseashta and S. Vaseashta. A Survey of Sensor Network Security. In *Sensors & Transducers Journal* Vol 94 Issue 7, 2008.

[6] U. Colesanti and S. Santini. A Performance Evaluation Of The Collection Tree Protocol Based On Its Implementation For The Castalia Wireless Sensor Networks Simulator. *Department of Computer Science, ETH Zurich* Tech Rep 681, 2010.

[7] O. Gnawali, et al. The Collection Tree Protocol. *http://sing.stanford.edu/gnawali/ctp/.*

[8] T. Shu, M. Krunz, and S. Liu. Secure data Collection in Wireless Sensor Networks Using Randomized Dispersive Routes. In *IEEE Transactions on Mobile Computing* Vol 9 No 7, 2010.

[9] G. Tel. *Introducion to Distributed Algorithms.* New York: Cambridge University Press, 2000.

[10] J. Deng, R. Han, and S. Mishra. INSENS: Intrusion–Tolerant Routing For Wireless Sensor Networks. *Computer Communications* 29, 2006.

[11] I. Krontiris, et al. Intrusion Detection of Sinkhole Attacks in Wireless Sensor Networks. *Anthens Information Technology*, 2007.

[12] D. Dolev and A. Yao. On the Security of Public Key Protocols. In *IEEE Transactions on Information Theory 29*(2) 1983.

[13] W. Pestman. *Mathematical Statistics: An Introduction*. Berlin: Walter De Gruyter Co., 1998.

[14] J. Gentle. *Elements of Computational Statistics,* New York: Springer-Verlag, 2002.

[15] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring The Internet* New York: Addison Wesley, 2005.

[16] "IRIS Datasheet." The IRIS Wireless Measurement System. *Crossbow Technology, Inc.*

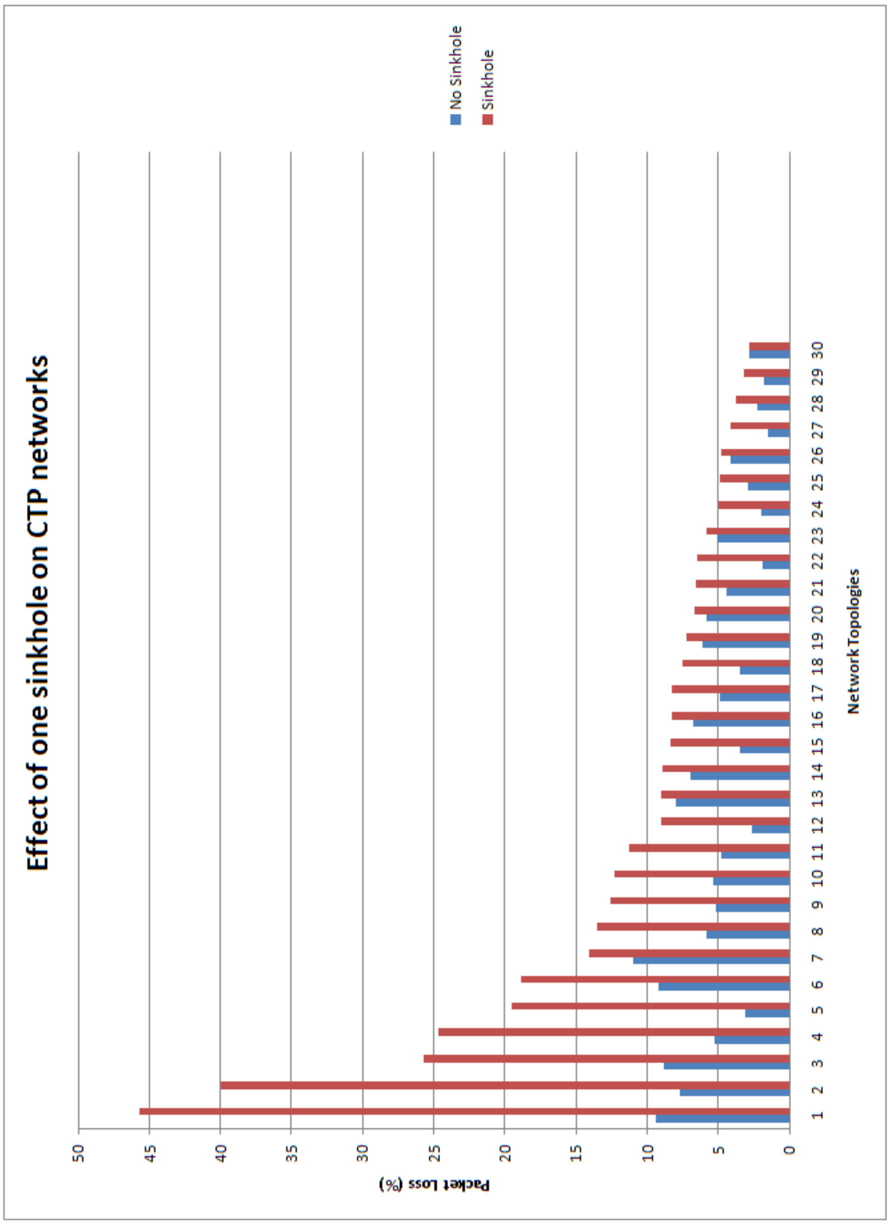
[17] "MICA2 Datasheet." The MICA2 Wireless Measurement System. *Crossbow Technology, Inc.*

[18] "TinyOS." *http://www.tinyos.net/*.

A Appendix

A.1 Effect of Sinkhole in Simulation

Note: This graph is described in the Simulations section



A.2 Source Code

CTP Routing Engine:

```
#include "CtpRoutingEngine.h"
#include "CtpForwardingEngine.h"
#include "LinkEstimator.h"
Define_Module(CtpRoutingEngine);

void CtpRoutingEngine::initialize(){
	//////////////////////// Castalia Implementation ///////////////////
	////////////////////////////////////////////////////////////////////

	//Pointers to other modules for direct function calls.
	cfe = check_and_cast<CtpForwardingEngine*>(getParentModule()->getSubmodule("CtpForwardingEngine"))
		;
	le = check_and_cast<LinkEstimator*>(getParentModule()->getSubmodule("LinkEstimator")) ;

	//Id of the node (like TOS_NODE_ID)
	self = getParentModule()->getParentModule()->getParentModule()->getIndex();

	// The default values are set in CtpRoutingEngine.ned
	// but they can be overwritten in omnetpp.ini
	routingTableSize = par("routingTableSize"); // default 10 entries
	minInterval = par("minInterval"); // default 128
	maxInterval = par("maxInterval") ; // default 512000
	ctpReHeaderSize = par("ctpReHeaderSize") ; // default header size: 5 bytes
	isRoot = par("isRoot") ; // sets this node as root


	// Clock drift simulation (it is present at each layer)
	if (getParentModule()->getParentModule()->getParentModule()->findSubmodule("ResourceManager") !=
		-1) {
		resMgrModule = check_and_cast <ResourceManager*>(getParentModule()->getParentModule()->
			getParentModule()->getSubmodule("ResourceManager"));
	} else {
		opp_error("\n[Mac]:\n Error in geting a valid reference to ResourceManager for direct 
			method calls.");
	}
	setTimerDrift(resMgrModule->getCPUClockDrift());

	////////////////////////////////////////////////////////////////////
	////////////////////////////////////////////////////////////////////

	//////////////////// CtpForwardingEngine (default) ////////////////
	////////////////////////////////////////////////////////////////////

	ECNOff = true;
	radioOn = true ; // TO IMPLEMENT ------ radioOn in stdcontrol
	running = false ;
	sending = false ;
	justEvicted = false ;

	routingTable = new routing_table_entry[routingTableSize] ;

	currentInterval = minInterval;

	////////////////////////////////////////////////////////////////////
	////////////////////////////////////////////////////////////////////

	////////////////////////// Init.init() /////////////////////////////
	////////////////////////////////////////////////////////////////////

	routeUpdateTimerCount = 0;
	parentChanges = 0;
	state_is_root = 0;

	routeInfoInit(&routeInfo);
```

```
        routingTableInit();
        my_ll_addr = command_AMPacket_address() ;

        beaconMsg = new CtpBeacon() ;
        beaconMsg->setByteLength(ctpReHeaderSize) ;

        // Call the corresponding rootcontrol command
        isRoot? command_RootControl_setRoot() : command_RootControl_unsetRoot() ;

        ////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////


        //////////////////////////////// Statistics ////////////////////////////////
        ////////////////////////////////////////////////////////////////////////////

        declareOutput("Ctp_Beacons") ;

        ////////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////////
}

void CtpRoutingEngine::handleMessage(cMessage* msg){
        int msgKind = msg->getKind();
                switch (msgKind) {
        case TIMER_SERVICE:{
                handleTimerMessage(msg);
                break;
        }
        default:{
                opp_error("Unkown_message_type.") ;
        }
                }
                delete msg ;

}

void CtpRoutingEngine::timerFiredCallback(int timer)
{
        trace() << "CtpRE_-_TimerFiredCallback,_value:_"<<timer;
        switch (timer) {

                case ROUTE_TIMER:{
                        setTimer(ROUTE_TIMER,tosMillisToSeconds(BEACON_INTERVAL)) ; // because it's a
                             periodic timer.
                        event_RouteTimer_fired() ;
                        break;
                }
                case BEACON_TIMER:{
                        event_BeaconTimer_fired() ;
                        break;
                }
                case POST_UPDATEROUTETASK:{
                        updateRouteTask() ;
                        break ;
                }

                case POST_SENDBEACONTASK:{
                        sendBeaconTask() ;
                        break ;
                }


                default:{
                        opp_error("Unexpected_message!");
                }
        }
}

CtpRoutingEngine::~CtpRoutingEngine(){
```

```
        delete beaconMsg ;
        beaconMsg = NULL ;

        delete [] routingTable ;
        routingTable = NULL ;
}

void CtpRoutingEngine::chooseAdvertiseTime() {
        t = currentInterval;
        t /= 2;
        t += command_Random_rand32(1) % t;
        tHasPassed = false;
        setTimer(BEACON_TIMER, tosMillisToSeconds(t)) ;
}

void CtpRoutingEngine::resetInterval() {
  currentInterval = minInterval;
  chooseAdvertiseTime();
}

void CtpRoutingEngine::decayInterval() {
        currentInterval *= 2;
        if (currentInterval > maxInterval) {
                currentInterval = maxInterval;
        }
        chooseAdvertiseTime();
}

void CtpRoutingEngine::remainingInterval() {
   uint32_t remaining = currentInterval;
   remaining -= t;
   tHasPassed = true;
   setTimer(BEACON_TIMER, tosMillisToSeconds(remaining)) ;
}

error_t CtpRoutingEngine::command_StdControl_start() {
        Enter_Method("command_StdControl_start") ;
        //start will (re)start the sending of messages
        if (!running) {
                running = true;
                resetInterval();
                setTimer(ROUTE_TIMER, tosMillisToSeconds(BEACON_INTERVAL)) ;
                trace()<<"stdControl.start - running "<<running<<" radioOn: "<<radioOn ;
        }
        return SUCCESS;
}

error_t CtpRoutingEngine::command_StdControl_stop() {
        Enter_Method("command_StdControl_stop") ;
        running = false ;
        trace()<<"stdControl.stop - running "<<running<<" radioOn: "<<radioOn ;
        return SUCCESS;
}

void CtpRoutingEngine::event_RadioControl_startDone(error_t error) {
        Enter_Method("event_RadioControl_startDone") ;
        radioOn = true;
        trace()<<"radioControl.startDone - running "<<running<<" radioOn: "<<radioOn ;
        if (running) {
                uint16_t nextInt;
                nextInt = command_Random_rand16(0) % BEACON_INTERVAL ;
                nextInt += BEACON_INTERVAL >> 1;
                setTimer(BEACON_TIMER, tosMillisToSeconds(nextInt));
        }
}

void CtpRoutingEngine::event_RadioControl_stopDone(error_t error) {
        Enter_Method("event_RadioControl_stopDone") ;
        radioOn = false;
        trace()<<"radioControl.stopDone - running "<<running<<" radioOn: "<<radioOn ;
```

```

}

/* Is this quality measure better than the minimum threshold? */
// Implemented assuming quality is EETX
bool CtpRoutingEngine::passLinkEtxThreshold(uint16_t etx) {
        return true;
        //      return (etx < ETX_THRESHOLD);
}

/* Converts the output of the link estimator to path metric
 * units, that can be *added* to form path metric measures */
uint16_t CtpRoutingEngine::evaluateEtx(uint16_t quality) {
        trace()<<"evaluateEtx - "<<(int) quality<<" -> "<<(int)(quality+10);
        return (quality + 10);
}


/* updates the routing information, using the info that has been received
 * from neighbor beacons. Two things can cause this info to change:
 * neighbor beacons, changes in link estimates, including neighbor eviction */
void CtpRoutingEngine::updateRouteTask() {
        uint8_t i;
        routing_table_entry* entry;
        routing_table_entry* best;
        uint16_t minEtx;
        uint16_t currentEtx;
        uint16_t linkEtx, pathEtx;

        if (state_is_root)
                return;

        best = NULL;
        /* Minimum etx found among neighbors, initially infinity */
        minEtx = MAX_METRIC;
        /* Metric through current parent, initially infinity */
        currentEtx = MAX_METRIC;

        trace()<<"updateRouteTask" ;

        /* Find best path in table, other than our current */
        for (i = 0; i < routingTableActive; i++) {
                entry = &routingTable[i];

                // Avoid bad entries and 1-hop loops
                if (entry->info.parent == INVALID_ADDR || entry->info.parent == my_ll_addr) {
                        trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "<<(int)entry->neighbor<<" parent: "<<entry->info.parent<<"  etx: NO_ROUTE]" ;
                        continue;
                }
                /* Compute this neighbor's path metric */
                linkEtx = evaluateEtx(/*call LinkEstimator.getLinkQuality(entry->neighbor) changed with: */ le->command_LinkEstimator_getLinkQuality(entry->neighbor));
                trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "<<(int)entry->neighbor<<" parent: "<<entry->info.parent<<"  etx: "<<(int)linkEtx
                  <<" pstId: "<<(int)entry->nonepstId<<"]";
                pathEtx = linkEtx + entry->info.etx;
                /* Operations specific to the current parent */
                if (entry->neighbor == routeInfo.parent) {
                        trace()<<"already parent";
                        currentEtx = pathEtx;
                        /* update routeInfo with parent's current info */
                        routeInfo.etx = entry->info.etx;
                        routeInfo.congested = entry->info.congested;
                        continue;
                }
                /* Ignore links that are congested */
                if (entry->info.congested)
                        continue;
                /* Ignore links that are bad */
```

```
                if (!passLinkEtxThreshold(linkEtx)) {
                        trace()<<"did not pass threshold.";
                        continue;
                }

                if (pathEtx < minEtx) {
                        minEtx = pathEtx;
                        best = entry;
                }
        }


        /* Now choose between the current parent and the best neighbor */
        /* Requires that:
            1. at least another neighbor was found with ok quality and not congested
            2. the current parent is congested and the other best route is at least as good
            3. or the current parent is not congested and the neighbor quality is better by
               the PARENT_SWITCH_THRESHOLD.
          Note: if our parent is congested, in order to avoid forming loops, we try to select
                a node which is not a descendent of our parent. routeInfo.ext is our parent's
                etx. Any descendent will be at least that + 10 (1 hop), so we restrict the
                selection to be less than that.
         */
        if (minEtx != MAX_METRIC) {
                if (currentEtx == MAX_METRIC ||
                                (routeInfo.congested && (minEtx < (routeInfo.etx + 10))) ||
                                minEtx + PARENT_SWITCH_THRESHOLD < currentEtx) {
                        // routeInfo.metric will not store the composed metric.
                        // since the linkMetric may change, we will compose whenever
                        // we need it: i. when choosing a parent (here);
                        //            ii. when choosing a next hop
                        parentChanges++;

                        trace()<<"Changed parent. from "<<(int)routeInfo.parent<<" to "<<(int)best->
                            neighbor<<" with pstId "<<(int)best->nonepstId;
                        le->command_LinkEstimator_unpinNeighbor(routeInfo.parent) ;
                        le->command_LinkEstimator_pinNeighbor(best->neighbor) ;
                        le->command_LinkEstimator_clearDLQ(best->neighbor) ;

                        routeInfo.parent = best->neighbor;
                        routeInfo.etx = best->info.etx;
                        routeInfo.congested = best->info.congested;
                        nonemy_pstId = best->nonepstId;
                }
        }

        /* Finally, tell people what happened:  */
        /* We can only loose a route to a parent if it has been evicted. If it hasn't
         * been just evicted then we already did not have a route */
        if (justEvicted && routeInfo.parent == INVALID_ADDR)
                signal_Routing_noRoute() ;
        /* On the other hand, if we didn't have a parent (no currentEtx) and now we
         * do, then we signal route found. The exception is if we just evicted the
         * parent and immediately found a replacement route: we don't signal in this
         * case */
        else if (!justEvicted &&
                        currentEtx == MAX_METRIC &&
                        minEtx != MAX_METRIC)
                signal_Routing_routeFound() ;
        justEvicted = false;
}

/* send a beacon advertising this node's routeInfo */
// only posted if running and radioOn
void CtpRoutingEngine::sendBeaconTask() {
        error_t eval;
        if (sending) {
                return;
        }

```

```
        beaconMsg->setOptions(0);

        /* Congestion notification: am I congested? */
        if (cfe->command_CtpCongestion_isCongested()) {
                beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_ECN);
        }

        beaconMsg->setParent(routeInfo.parent);
        if (state_is_root) {
                beaconMsg->setEtx(routeInfo.etx);
        }
        else if (routeInfo.parent == INVALID_ADDR) {
                beaconMsg->setEtx(routeInfo.etx);
                beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_PULL);
        } else {
                beaconMsg->setEtx(routeInfo.etx + le->command_LinkEstimator_getLinkQuality(routeInfo.parent));
        }

        beaconMsg->nonesetPstId(nonemy_pstId); // hmm; i should have set my_pstId to that of my parent

        trace()<<"sendBeaconTask - parent: "<<(int)beaconMsg->getParent()<<" etx: "<<(int)beaconMsg->getEtx();

        beaconMsg->getRoutingInteractionControl().lastHop = self; // ok
        eval = le->command_Send_send(AM_BROADCAST_ADDR, beaconMsg->dup()); // the duplicate will be deleted in the LE module, we keep a copy here that is reused each time.

        if (eval == SUCCESS) {
                //statistics
                collectOutput("Ctp_Beacons","Tx");
                sending = true;
        } else if (eval == EOFF) {
                radioOn = false;
                trace()<<"sendBeaconTask - running: "<<running<<" radioOn: "<<radioOn;
        }
}

void CtpRoutingEngine::event_BeaconSend_sendDone(cMessage* msg, error_t error) {
        Enter_Method("event_BeaconSend_sendDone");
        if (!sending) {
                //something smells bad around here
                opp_error("something smells bad around here");
                return;
        }
        sending = false;
}

void CtpRoutingEngine::event_RouteTimer_fired() {
        if (radioOn && running) {
                post_updateRouteTask();
        }
}

void CtpRoutingEngine::event_BeaconTimer_fired() {
        if (radioOn && running) {
                if (!tHasPassed) {
                        post_updateRouteTask(); // always the most up to date info
                        post_sendBeaconTask();
                        trace()<<"Beacon timer fired.";
                        remainingInterval();
                }
                else {
                        decayInterval();
                }
        }
}

/*
 * We don't need a pointer to the header, we can use the methods of cPacket
```

```
 * instead, we return a pointer to CtpBeacon and that's it.
 */
CtpBeacon* CtpRoutingEngine::getHeader(cPacket* msg){
        return check_and_cast<CtpBeacon*>(msg) ;
}

/* Handle the receiving of beacon messages from the neighbors. We update the
 * table, but wait for the next route update to choose a new parent */
void CtpRoutingEngine::event_BeaconReceive_receive(cPacket* msg) {
        Enter_Method("event_BeaconReceive_receive") ;
        am_addr_t from;
        bool congested;

        //statistics
        collectOutput("Ctp Beacons","Rx") ;

        // we skip the check of beacon length.

        //need to get the am_addr_t of the source
        from = command_AMPacket_source(msg) ;
        CtpBeacon* rcvBeacon = check_and_cast<CtpBeacon*>(msg) ;

        congested = command_CtpRoutingPacket_getOption(msg,CTP_OPT_ECN) ;

        trace()<<"BeaconReceive.receive - from "<<(int)from<<" [parent: "<<(int)rcvBeacon->getParent()<<" etx: "<<(int)rcvBeacon->getEtx()
            <<" pstId: "<<(int)rcvBeacon->nonegetPstId()<<"]";
        //update neighbor table
        if (rcvBeacon->getParent() != INVALID_ADDR) {

                /* If this node is a root, request a forced insert in the link
                 * estimator table and pin the node. */
                if (rcvBeacon->getEtx() == 0) {
                        trace()<<"from a root, inserting if not in table "<<"my_ll_addr: "<<my_ll_addr;
                        le->command_LinkEstimator_insertNeighbor(from) ;
                        le->command_LinkEstimator_pinNeighbor(from) ;
                        nonemy_pstId = my_ll_addr;  // since i hear root, i'm the root of a principle subtree
                }
                //TODO: also, if better than my current parent's path etx, insert

                if (rcvBeacon->getEtx() == 0)
                        routingTableUpdateEntry(from, rcvBeacon->getParent(), rcvBeacon->getEtx(), nonemy_pstId);
                else
                        routingTableUpdateEntry(from, rcvBeacon->getParent(), rcvBeacon->getEtx(), rcvBeacon->nonegetPstId());

                command_CtpInfo_setNeighborCongested(from,congested) ;
        }

        if (command_CtpRoutingPacket_getOption(msg, CTP_OPT_PULL)) {
                resetInterval();
        }
        delete msg ;
        // we do not return the message, we delete it.
}

/* Signals that a neighbor is no longer reachable. need special care if
 * that neighbor is our parent */
void CtpRoutingEngine::event_LinkEstimator_evicted(am_addr_t neighbor) {
        Enter_Method("event_LinkEstimator_evicted") ;
        routingTableEvict(neighbor);
        trace()<<"LinkEstimator.evicted" ;
        if (routeInfo.parent == neighbor) {
                routeInfoInit(&routeInfo);
                justEvicted = true;
                post_updateRouteTask() ;
        }
}
```

```

/*
 * UnicastNameFreeRouting Inteface ----------------------------------------
 */
/* Simple implementation: return the current routeInfo */
am_addr_t CtpRoutingEngine::command_Routing_nextHop() {
	Enter_Method("command_Routing_nextHop") ;
	return routeInfo.parent;
}
bool CtpRoutingEngine::command_Routing_hasRoute() {
	Enter_Method("command_Routing_hasRoute") ;
	return (routeInfo.parent != INVALID_ADDR);
}
// ----------------------------------------------------------------------------

/*
 * CtpInfo Interface (Part 1) ---------------------------------------------------------
 */
error_t CtpRoutingEngine::command_CtpInfo_getParent(am_addr_t* parent) {
	if (parent == NULL)
		return FAIL;
	if (routeInfo.parent == INVALID_ADDR)
		return FAIL;
	*parent = routeInfo.parent;
	return SUCCESS;
}

error_t CtpRoutingEngine::command_CtpInfo_getEtx(uint16_t* etx) {
	Enter_Method("command_CtpInfo_getEtx") ;
	if (etx == NULL)
		return FAIL;
	if (routeInfo.parent == INVALID_ADDR)
		return FAIL;
	if (state_is_root == 1) {
		*etx = 0;
	} else {
		// path etx = etx(parent) + etx(link to the parent)
		*etx = routeInfo.etx + evaluateEtx(le->command_LinkEstimator_getLinkQuality(routeInfo.
			parent) );
	}
	return SUCCESS;
}

void CtpRoutingEngine::command_CtpInfo_recomputeRoutes() {
	Enter_Method("command_CtpInfo_recomputeRoutes") ;
	post_updateRouteTask() ;
}

void CtpRoutingEngine::command_CtpInfo_triggerRouteUpdate() {
	Enter_Method("command_CtpInfo_triggerRouteUpdate") ;
	resetInterval();
}

void CtpRoutingEngine::command_CtpInfo_triggerImmediateRouteUpdate() {
	Enter_Method("command_CtpInfo_triggerImmediateRouteUpdate") ;
	resetInterval();
}

void CtpRoutingEngine::command_CtpInfo_setNeighborCongested(am_addr_t n, bool congested) {
	Enter_Method("command_CtpInfo_setNeighborCongested") ;
	uint8_t idx;
	if (ECNOff)
		return;
	idx = routingTableFind(n);
	if (idx < routingTableActive) {
		routingTable[idx].info.congested = congested;
	}
	if (routeInfo.congested && !congested)
		post_updateRouteTask() ;
	else if (routeInfo.parent == n && congested)
```

```
                post_updateRouteTask() ;
}

bool CtpRoutingEngine::command_CtpInfo_isNeighborCongested(am_addr_t n) {
        Enter_Method("command_CtpInfo_isNeighborCongested") ;
        uint8_t idx;

        if (ECNOff)
                return false;

        idx = routingTableFind(n);
        if (idx < routingTableActive) {
                return routingTable[idx].info.congested;
        }
        return false;
}
// ----------------------------------------------------------------------------

/*
 *  RootControl Interface -----------------------------------------------------
 */
/** sets the current node as a root, if not already a root */
/*  returns FAIL if it's not possible for some reason      */
error_t CtpRoutingEngine::command_RootControl_setRoot() {
        Enter_Method("command_RootControl_setRoot") ;
        bool route_found = false ;
        route_found = (routeInfo.parent == INVALID_ADDR);
        state_is_root = 1;
        nonemy_pstId  = 0; // a root is no principle subtree
        routeInfo.parent = my_ll_addr; //myself
        routeInfo.etx = 0;
        if (route_found)
                signal_Routing_routeFound() ;
        trace()<<"RootControl.setRoot - I'm a root now!"<<(int) routeInfo.parent  ;
        return SUCCESS;
}

error_t CtpRoutingEngine::command_RootControl_unsetRoot() {
        Enter_Method("command_RootControl_unsetRoot") ;
        state_is_root = 0;
        routeInfoInit(&routeInfo);
        trace()<<"RootControl.unsetRoot - I'm not a root now!" ;
        post_updateRouteTask() ;
        return SUCCESS;
}

bool CtpRoutingEngine::command_RootControl_isRoot() {
        Enter_Method("command_RootControl_isRoot") ;
        return state_is_root;
}
// ----------------------------------------------------------------------------

// default events Routing.noRoute and Routing.routeFound skipped -> useless

/* This should see if the node should be inserted in the table.
 * If the white_bit is set, this means the LL believes this is a good
 * first hop link.
 * The link will be recommended for insertion if it is better* than some
 * link in the routing table that is not our parent.
 * We are comparing the path quality up to the node, and ignoring the link
 * quality from us to the node. This is because of a couple of things:
 *   1. because of the white bit, we assume that the 1-hop to the candidate
 *      link is good (say, etx=1)
 *   2. we are being optimistic to the nodes in the table, by ignoring the
 *      1-hop quality to them (which means we are assuming it's 1 as well)
 *      This actually sets the bar a little higher for replacement
 *   3. this is faster
 *   4. it doesn't require the link estimator to have stabilized on a link
 */
bool CtpRoutingEngine::event_CompareBit_shouldInsert(cPacket *msg, bool white_bit) {
```

```
        Enter_Method("event_CompareBit_shouldInsert") ;
        bool found = false;
        uint16_t pathEtx;
        uint16_t neighEtx;
        int i;
        routing_table_entry* entry;
        CtpBeacon* rcvBeacon ;

        // checks if it is a CtpBeacon
        if(dynamic_cast<CtpBeacon*>(msg) == NULL){
                delete msg ;
                return false ;
        }

        /* 1.determine this packet's path quality */
        rcvBeacon = check_and_cast<CtpBeacon*>(msg); // we don't need a pointer to header, we use cPacket
            methods.

        if (rcvBeacon->getParent() == INVALID_ADDR){
                delete msg ;
                return false;
        }

        /* the node is a root, recommend insertion! */
        if (rcvBeacon->getEtx() == 0) {
                delete msg ;
                return true;
        }

        pathEtx = rcvBeacon->getEtx() ;

        /* 2. see if we find some neighbor that is worse */
        for (i = 0; i < routingTableActive && !found; i++) {
                entry = &routingTable[i];
                //ignore parent, since we can't replace it
                if (entry->neighbor == routeInfo.parent)
                        continue;
                neighEtx = entry->info.etx;
                //neighEtx = evaluateEtx(call LinkEstimator.getLinkQuality(entry->neighbor));
                found |= (pathEtx < neighEtx);
        }
        delete msg ;
        return found;
}

/**********************************************************/
/* Routing Table Functions                                */

/* The routing table keeps info about neighbor's route_info,
 * and is used when choosing a parent.
 * The table is simple:
 *   - not fragmented (all entries in 0..routingTableActive)
 *   - not ordered
 *   - no replacement: eviction follows the LinkEstimator table
 */

void CtpRoutingEngine::routingTableInit() {
        routingTableActive = 0;
}

/* Returns the index of parent in the table or
 * routingTableActive if not found */
uint8_t CtpRoutingEngine::routingTableFind(am_addr_t neighbor) {
        uint8_t i;
        if (neighbor == INVALID_ADDR)
                return routingTableActive;
        for (i = 0; i < routingTableActive; i++) {
                if (routingTable[i].neighbor == neighbor)
                        break;
        }
```

```
        return i;
}


error_t CtpRoutingEngine::routingTableUpdateEntry(am_addr_t from, am_addr_t parent, uint16_t etx, am_addr_t nonepstId)    {
        uint8_t idx;
        uint16_t  linkEtx;
        linkEtx = evaluateEtx(le->command_LinkEstimator_getLinkQuality(from));

        idx = routingTableFind(from);
        if (idx == routingTableSize) {
                //not found and table is full
                //if (passLinkEtxThreshold(linkEtx))
                //TODO: add replacement here, replace the worst
                //}
                trace()<<"routingTableUpdateEntry - FAIL, table full";
                return FAIL;
        }
        else if (idx == routingTableActive) {
                //not found and there is space
                if (passLinkEtxThreshold(linkEtx)) {
                        routingTable[idx].neighbor = from;
                        routingTable[idx].nonepstId = nonepstId;
                        routingTable[idx].info.parent = parent;
                        routingTable[idx].info.etx = etx;
                        routingTable[idx].info.haveHeard = 1;
                        routingTable[idx].info.congested = false;
                        routingTableActive++;
                        trace()<<"routingTableUpdateEntry - OK, new entry";
                        if (nonepstId != nonemy_pstId)
                                trace()<<"## my_pstId: "<<(int)nonemy_pstId<<" neighbor_pstId: "<<nonepstId;
                } else {
                        trace()<<"routingTableUpdateEntry - Fail, link quality ("<<(int)linkEtx<<") below threshold" ;
                }
        } else {
                //found, just update
                routingTable[idx].neighbor = from;
                routingTable[idx].info.parent = parent;
                routingTable[idx].info.etx = etx;
                routingTable[idx].info.haveHeard = 1;
                trace()<<"routingTableUpdateEntry - OK, updated entry";
        }
        return SUCCESS;
}

/* if this gets expensive, introduce indirection through an array of pointers */
error_t CtpRoutingEngine::routingTableEvict(am_addr_t neighbor) {
        uint8_t idx,i;
        idx = routingTableFind(neighbor);
        if (idx == routingTableActive)
                return FAIL;
        routingTableActive--;
        for (i = idx; i < routingTableActive; i++) {
                routingTable[i] = routingTable[i+1];
        }
        return SUCCESS;
}
/*********** end routing table functions ***************/

// Collection Debug skipped -> not useful in our implementation

/*
 * CtpRoutingPacket Interface ---------------------------------------------------------------
 */
bool CtpRoutingEngine::command_CtpRoutingPacket_getOption(cPacket* msg, ctp_options_t opt) {
        return ((getHeader(msg)->getOptions() & opt) == opt) ? true : false;
}
```

```
void CtpRoutingEngine::command_CtpRoutingPacket_setOption(cPacket* msg, ctp_options_t opt) {
        getHeader(msg)->setOptions(getHeader(msg)->getOptions() | opt) ;
}

void CtpRoutingEngine::command_CtpRoutingPacket_clearOption(cPacket* msg, ctp_options_t opt) {
        getHeader(msg)->setOptions(getHeader(msg)->getOptions() & ~opt) ;
}

void CtpRoutingEngine::command_CtpRoutingPacket_clearOptions(cPacket* msg) {
        getHeader(msg)->setOptions(0) ;
}

am_addr_t CtpRoutingEngine::command_CtpRoutingPacket_getParent(cPacket* msg) {
        return getHeader(msg)->getParent();
}
void CtpRoutingEngine::command_CtpRoutingPacket_setParent(cPacket* msg, am_addr_t addr) {
        getHeader(msg)->setParent(addr);
}

uint16_t CtpRoutingEngine::command_CtpRoutingPacket_getEtx(cPacket* msg) {
        return getHeader(msg)->getEtx();
}

void CtpRoutingEngine::command_CtpRoutingPacket_setEtx(cPacket* msg, uint8_t etx) {
        getHeader(msg)->setEtx(etx);
}
// ---------------------------------------------------------------------------------------------------

/*
 * CtpInfo Interface (Part 2) ---------------------------------------------------------------
 */
uint8_t CtpRoutingEngine::command_CtpInfo_numNeighbors() {
        return routingTableActive;
}

uint16_t CtpRoutingEngine::command_CtpInfo_getNeighborLinkQuality(uint8_t n) {
        return (n < routingTableActive)? le->command_LinkEstimator_getLinkQuality(routingTable[n].neighbor
            ):0xffff;
}

uint16_t CtpRoutingEngine::command_CtpInfo_getNeighborRouteQuality(uint8_t n) {
        return (n < routingTableActive)? le->command_LinkEstimator_getLinkQuality(routingTable[n].neighbor
            ) + routingTable[n].info.etx:0xfffff;
}

am_addr_t CtpRoutingEngine::command_CtpInfo_getNeighborAddr(uint8_t n) {
        return (n < routingTableActive)? routingTable[n].neighbor:AM_BROADCAST_ADDR ;
}
// ---------------------------------------------------------------------------------------------------


//////////////////////////// Custom functions /////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////

// These functions trigger an event in the module where they should signal it.
void CtpRoutingEngine::signal_Routing_routeFound(){
        cfe->event_UnicastNameFreeRouting_routeFound() ;
}
void CtpRoutingEngine::signal_Routing_noRoute(){
        cfe->event_UnicastNameFreeRouting_routeFound() ;
}

/*
 * AMPacket Interface (just what we need) --------------------------------
 */
am_addr_t CtpRoutingEngine::command_AMPacket_source(cMessage* msg){
        RoutingPacket* rPkt = check_and_cast<RoutingPacket*>(msg) ;
        return (uint16_t) rPkt->getRoutingInteractionControl().lastHop ;
}
```

```
am_addr_t CtpRoutingEngine::command_AMPacket_address(){
        return self ;
}
// ----------------------------------------------------------------------------

// these functions simulate the post command of TinyOs
void CtpRoutingEngine::post_updateRouteTask(){
        setTimer(POST_UPDATEROUTETASK,0); // cannot call the updateRouteTask directly. By this way it is
             more similar to the post command in TinyOs.
}

void CtpRoutingEngine::post_sendBeaconTask(){
        setTimer(POST_SENDBEACONTASK,0) ;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

CTP Routing Engine Header:

```
#ifndef _CTPROUTINGENGINE_H_
#define _CTPROUTINGENGINE_H_

#include "Ctp.h"

using namespace std;


/////////////////////////// TreeRouting.h ///////////////////////////
/////////////////////////////////////////////////////////////////////

enum {
        AM_TREE_ROUTING_CONTROL = 0xCE,
        BEACON_INTERVAL = 8192,
        INVALID_ADDR  = 0xffff ,
        ETX_THRESHOLD = 50,       // link quality=20% -> ETX=5 -> Metric=50
        PARENT_SWITCH_THRESHOLD = 15,
        MAX_METRIC = 0xFFFF,
};


typedef struct {
        am_addr_t parent;
        uint16_t etx;
        bool haveHeard;
        bool congested;
} route_info_t;

typedef struct {
        am_addr_t neighbor;
        am_addr_t nonepstId;
        route_info_t info;
} routing_table_entry;

inline void routeInfoInit(route_info_t *ri) {
        ri->parent = INVALID_ADDR;
        ri->etx = 0;
        ri->haveHeard = 0;
        ri->congested = false ;
}

/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////


////////////////////////// Custom values ///////////////////////////
/////////////////////////////////////////////////////////////////////
```

```

enum{
        BEACON_TIMER = 1,
        ROUTE_TIMER = 2,
        POST_UPDATEROUTETASK = 3,
        POST_SENDBEACONTASK = 4,
};

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

class CtpRoutingEngine ;
class LinkEstimator ;
class CtpRoutingEngine: public CastaliaModule , public TimerService{
 protected:

        /////////////////////// CtpRoutingEngineP.nc /////////////////////////
        ////////////////////////////////////////////////////////////////////////

        bool ECNOff ;
        bool radioOn ;
        bool running ;
        bool sending ;
        bool justEvicted ;

        route_info_t routeInfo ;
        bool state_is_root;
        am_addr_t my_ll_addr;

        am_addr_t nonemy_pstId; // my pstID is that of my parent'

        cPacket beaconMsgBuffer ;
        CtpBeacon* beaconMsg ; // we don't need a pointer to the header, we use methods of cPacket instead
            .

        /* routing table -- routing info about neighbors */
        routing_table_entry* routingTable ;
        uint8_t routingTableActive;

        /* statistics */
        uint32_t parentChanges;
        /* end statistics */

        uint32_t routeUpdateTimerCount;

        uint32_t currentInterval ;
        uint32_t t;
        bool tHasPassed;

        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////

        ////////////////////// Custom Variables /////////////////////////////
        ////////////////////////////////////////////////////////////////////////

        // Pointers to other modules.
        CtpForwardingEngine *cfe ;
        LinkEstimator *le ;
        ResourceManager* resMgrModule;

        // Beacon Frame size.
        int ctpReHeaderSize ;

        // Node Id.
        int self;

        // Sets a node as root from omnetpp.ini
        bool isRoot ;

        // Sets a node as a sinkhole from omnetpp.ini
```

```
        bool isSink;

        // Arguments of generic module CtpRoutingEngineP
        uint32_t minInterval ;
        uint32_t maxInterval ;
        uint8_t routingTableSize ;

        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////


        ///////////////// Castalia Functions /////////////////////////////
        ///////////////////////////////////////////////////////////////////

        virtual void initialize();
        virtual void handleMessage(cMessage* msg) ;
        void timerFiredCallback(int timer) ;
        virtual ~CtpRoutingEngine() ;

        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////

        ////////////////// CtpRoutingEngine functions ////////////////////
        ///////////////////////////////////////////////////////////////////

        void chooseAdvertiseTime();
        void resetInterval();
        void decayInterval();
        void remainingInterval() ;

        bool passLinkEtxThreshold(uint16_t etx) ;
        uint16_t evaluateEtx(uint16_t quality) ;

        void updateRouteTask() ;
        void sendBeaconTask() ;

        void event_RouteTimer_fired() ;
        void event_BeaconTimer_fired() ;

        CtpBeacon* getHeader(cPacket* msg) ;

        void routingTableInit() ;
        uint8_t routingTableFind(am_addr_t) ;
        error_t routingTableUpdateEntry(am_addr_t, am_addr_t, uint16_t, am_addr_t);
        error_t routingTableEvict(am_addr_t);

        // CtpRoutingPacket Interface ---------------------------------------------
        bool command_CtpRoutingPacket_getOption(cPacket* msg, ctp_options_t opt);
        void command_CtpRoutingPacket_setOption(cPacket* msg, ctp_options_t opt);
        void command_CtpRoutingPacket_clearOption(cPacket* msg, ctp_options_t opt);
        void command_CtpRoutingPacket_clearOptions(cPacket* msg);
        am_addr_t command_CtpRoutingPacket_getParent(cPacket* msg);
        void command_CtpRoutingPacket_setParent(cPacket* msg, am_addr_t addr);
        uint16_t command_CtpRoutingPacket_getEtx(cPacket* msg);
        void command_CtpRoutingPacket_setEtx(cPacket* msg, uint8_t etx);
        // -------------------------------------------------------------------------

        ///////////////////////////////////////////////////////////////////
        ///////////////////////////////////////////////////////////////////


        ////////////////// Custom functions //////////////////////////////
        ///////////////////////////////////////////////////////////////////

        // generates an event in the module where they should signal it.
        void signal_Routing_routeFound() ;
        void signal_Routing_noRoute() ;

        // AMPacket Interface (just what we need) ---------------------
        am_addr_t command_AMPacket_source(cMessage* msg);
```

```
        am_addr_t command_AMPacket_address();
        // -----------------------------------------------------------------

        // functions that simulate the post command of TinyOs
        void post_sendBeaconTask() ;
        void post_updateRouteTask() ;

        ////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////


 public:

        /////////////////// CtpRoutingEngine functions ////////////////////
        ////////////////////////////////////////////////////////////////////

        error_t command_StdControl_start();
        error_t command_StdControl_stop() ;
        void event_RadioControl_startDone(error_t error) ;
        void event_RadioControl_stopDone(error_t error) ;

        void event_BeaconSend_sendDone(cMessage* msg, error_t error) ;

        void event_BeaconReceive_receive(cPacket* msg);

        void event_LinkEstimator_evicted(am_addr_t neighbor) ;

        // UnicastNameFreeRouting Interface --------------------------------
        bool command_Routing_hasRoute() ;
        am_addr_t command_Routing_nextHop() ;
        // -----------------------------------------------------------------

        // CtpInfo Interface (Part 1) -------------------------------------------
        error_t command_CtpInfo_getParent(am_addr_t*) ;
        uint8_t command_CtpInfo_getEtx(uint16_t* etx) ;
        void command_CtpInfo_recomputeRoutes() ;
        void command_CtpInfo_triggerRouteUpdate() ;
        void command_CtpInfo_triggerImmediateRouteUpdate() ;
        void command_CtpInfo_setNeighborCongested(uint16_t n,bool congested) ;
        bool command_CtpInfo_isNeighborCongested(uint16_t addr) ;
        // ----------------------------------------------------------------------

        // RootControl Interface ------------------------------------------------
        bool command_RootControl_isRoot();
        error_t command_RootControl_setRoot() ;
        error_t command_RootControl_unsetRoot() ;
        // ----------------------------------------------------------------------

        bool event_CompareBit_shouldInsert(cPacket *msg, bool white_bit) ;

        // CtpInfo Interface (Part 2) -------------------------------------------
        uint8_t command_CtpInfo_numNeighbors() ;
        uint16_t command_CtpInfo_getNeighborLinkQuality(uint8_t) ;
        uint16_t command_CtpInfo_getNeighborRouteQuality(uint8_t) ;
        am_addr_t command_CtpInfo_getNeighborAddr(uint8_t) ;
        // ----------------------------------------------------------------------

        ////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////

};
#endif
```